USENIX

5th Symposium on Operating Systems Design and Implementation

Boston, MA, USA, December 2002

conference

proceedings

5th Symposium on Operating Systems Design and Implementation (OSDI '02)

Boston, Massachusetts, USA
December 9–11, 2002

Sponsored by
The USENIX Association
Co-sponsored by **ACM SIGOPS**
In Cooperation with **IEEE TCOS**

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $30 for members and $40 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-931971-06-4

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association

Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)

Co-sponsored by ACM SIGOPS

In Cooperation with IEEE TCOS

December 9–11, 2002
Boston, Massachusetts, USA

Symposium Organizers

Program Chair
David Culler, *University of California, Berkeley*
Peter Druschel, *Rice University*

Program Committee
Eric Brewer, *University of California, Berkeley*
Miguel Castro, *Microsoft Research*
Carla Ellis, *Duke University*
Dawson Engler, *Stanford University*
Deborah Estrin, *University of California, Los Angeles/ISI*
Greg Ganger, *Carnegie Mellon University*
Jim Gray, *Microsoft Research*
Jay Lepreau, *University of Utah*
Robert Morris, *Massachusetts Institute of Technology*
Timothy Roscoe, *Intel Research Lab at Berkeley*
Chandu Thekkath, *Microsoft Research*
David Wetherall, *University of Washington*

Steering Committee
Michael B. Jones, *Microsoft Research*
Frans Kaashoek, *Massachusetts Institute of Technology*

The USENIX Association Staff

External Reviewers
Atul Adya
Sameer Ajmani
David Andersen
Darrell Anderson
Andrea Arpaci-Dusseau
Remzi Arpaci-Dusseau
Tuomas Aura
Godmar Back
Paul Barham
Bobby Bhattacharjee
David Black
Richard Black
Trevor Blackwell
Nikita Borisov
Thomas Bressoud
Nevil Brownlee
Philip Buonadonna
Andrew Campbell
John Carter
Alberto Cerpa
Surendar Chandra
Jeff Chase
Benjie Chen
Mike Chen
Peter Chen
Andy Chou
Brent Chun
Patrick Crowley
Mike Dahlin
David Dill
John Douceur
Eric Eide
Kevin Eustice
Ted Faber
Mike Feeley
Jason Flinn
Rick Floyd
Cédric Fournet
Michael Franklin
Ayalvadi Ganesh
Johannes Gehrke
Garth Gibson
Daniel Giffin
Thomer M. Gil
Steve Gribble
Dirk Grunwald
Richard Guy
Steven Hand
Timothy Harris
Chris Hawblitzel
Mike Hibler
Michael Hicks
Wilson Hsieh
Galen Hunt
Sitaram Iyer
Trent Jaeger
Paul Jardetzky
Brad Karp
Kimberly Keeton
Anne-Marie Kermarrec
Eddie Kohler
David Kotz
Michael Kozuch
Orran Krieger
Lakshman Krishnamurthy
John Kubiatowicz
Eren Kursun
Alvin Lebeck
Philip Levis
Hank Levy
Jinyang Li
Kai Li
Darrell Long
Ewing Lusk
Mesaac Makpangou
Rob Malan
Laurent Massoulié
Roy Maxion
David Mazières
Ethan Miller
Jeffrey Mogul
Richard Mortier
David Mosberger
Todd Mowry
Gilles Muller
Andrew Myers
David Nagle
Erich Nahum
Juan Navarro
Silvia Nittel
Brian Noble
Chris Olston
Vivek Pai
Parveen Patel
Larry Peterson
David Petrou
Ian Pratt
Calton Pu
Vijay Raghunathan
Sylvia Ratnasamy
John Regehr
Alastair Reid
Peter Reiher
Mike Reiter
Erik Riedel
Rodrigo Rodrigues
Michael Roe
Mendel Rosenblum
Antony Rowstron
Constantine Sapuntzakis
Mahadev Satyanarayanan
Dan Scales
Margo Seltzer
Srinivasan Seshan
Jonathan Shapiro
Marc Shapiro
Dan Simon
Emin Gun Sirer
Stephen Smalley
Alex Snoeren
Ankur Srivastava
Dave Stewart
Doug Terry
Amin Vahdat
Alistair Veitch
Geoff Voelker
Werner Vogels
James von Behren
David Wagner
Carl Waldspurger
Jonathan Walpole
Randolph Wang
Matt Welsh
Brian White
Ted Wobber
Alec Wolman
Fan Ye
Erez Zadok
Ellen Zegura
Beichuan Zhang
Ben Zhao
Lidong Zhou
Yuanyuan Zhou
Ben Zorn

OSDI '02: 5th Symposium on Operating Systems Design and Implementation

December 9–11, 2002
Boston, Massachusetts, USA

Index of Authors

Message from the Program Chair

OSDI '02 continues the conference's tradition of presenting the best innovative work in the systems software area, taking a broad view of what the area encompasses. We believe that this year's conference contains some of the most original, intriguing, and important work in the field today. OSDI '02 drew a highly competitive selection from a large collection of diverse, original work submitted by authors internationally to form a program of creative, well-developed papers opening up new and innovative areas—not merely incremental results building on prior work.

The program presents important results in a wide range of areas, including distributed storage systems, robust software construction, OS kernel innovations, sensor networks, virtual machine monitors, large-scale network simulation, resource management architectures for Internet services, peer-to-peer systems, network analysis, and migration of execution environments.

OSDI '02 received 150 submissions; all submissions were reviewed by several members of the program committee and selected external reviewers, comprising roughly 700 reviews. Program committee members did not participate in the review or discussion of any papers co-authored by themselves, members of their home institutions, former students, advisors, or recent collaborators. The program committee met at the Intel Berkeley research lab on July 30, 2002, to make the final program selection during an eleven-hour meeting.

We wish to thank the Program Committee for their dedication and hard work in reviewing an unexpectedly high number of submissions and for selecting an excellent technical program. We also thank the outside reviewers for lending their expertise on short notice; the OSDI '02 sponsors: USENIX, ACM SIGOPS and IEEE TCOS, HP Labs, Intel Research, and Microsoft Research, for their support; and the members of the USENIX staff for expertly managing the conference logistics, marketing, and proceedings publication. We also wish to extend a special thanks to Dirk Grunwald for setting up and letting us use his impressive review management system on very short notice.

Finally, we wish to thank all authors who submitted papers to OSDI '02. Without their contributions, there would be no OSDI '02.

David Culler, University of California, Berkeley
Peter Druschel, Rice University
Symposium Co-Chairs

FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment

Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken,
John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, Roger P. Wattenhofer

Microsoft Research, Redmond, WA 98052

{adya, bolosky, mcastro, gcermak, rchaiken, johndo, howell, lorch, theimer}@microsoft.com;
wattenhofer@inf.ethz.ch

Abstract

Farsite is a secure, scalable file system that logically functions as a centralized file server but is physically distributed among a set of untrusted computers. Farsite provides file availability and reliability through randomized replicated storage; it ensures the secrecy of file contents with cryptographic techniques; it maintains the integrity of file and directory data with a Byzantine-fault-tolerant protocol; it is designed to be scalable by using a distributed hint mechanism and delegation certificates for pathname translations; and it achieves good performance by locally caching file data, lazily propagating file updates, and varying the duration and granularity of content leases. We report on the design of Farsite and the lessons we have learned by implementing much of that design.

1. Introduction

This paper describes Farsite, a serverless distributed file system that logically functions as a centralized file server but whose physical realization is dispersed among a network of untrusted desktop workstations. Farsite is intended to provide both the benefits of a central file server (a shared namespace, location-transparent access, and reliable data storage) and the benefits of local desktop file systems (low cost, privacy from nosy sysadmins, and resistance to geographically localized faults). Farsite replaces the physical security of a server in a locked room with the virtual security of cryptography, randomized replication, and Byzantine fault-tolerance [8]. Farsite is designed to support typical desktop file-I/O workloads in academic and corporate environments, rather than the high-performance I/O of scientific applications or the large-scale write sharing of database applications. It requires minimal administrative effort to initially configure and practically no central administration to maintain. With a few notable exceptions (such as crash recovery and interaction between multiple Byzantine-fault-tolerant groups), nearly all of the design we describe has been implemented.

Traditionally, file service for workstations has been provided either by a local file system such as FFS [28] or by a remote server-based file system such as NFS [39] or AFS [21]. Server-based file systems provide a shared namespace among users, and they can offer greater file reliability than local file systems because of better maintained, superior quality, and more highly redundant components. Servers also afford greater physical security than personal workstations in offices.

However, server-based systems carry a direct cost in equipment, physical plant, and personnel beyond those already sunk into the desktop infrastructure commonly found in modern companies and institutions. A server requires a dedicated administrative staff, upon whose competence its reliability depends [19] and upon whose trustworthiness its security depends [47]. Physically centralized servers are vulnerable to geographically localized faults, and their store of increasingly sensitive and valuable information makes them attractive, concentrated targets for subversion and data theft, in contrast to the inherent decentralization of desktop workstations.

In designing Farsite, our goal has been to harness the collective resources of loosely coupled, insecure, and unreliable machines to provide logically centralized, secure, and reliable file-storage service. Our system protects and preserves file data and directory metadata primarily through the techniques of cryptography and replication. Since file data is large and opaque to the system, the techniques of encryption, one-way hashing, and raw replication provide means to ensure its privacy, integrity, and durability, respectively. By contrast, directory metadata is relatively small, but it must be comprehensible and revisable directly by the system; therefore, it is maintained by Byzantine-replicated state-machines [8, 36] and specialized cryptographic techniques that permit metadata syntax enforcement without compromising privacy [15]. One of Farsite's key design objectives is to provide the benefits of Byzantine fault-tolerance while avoiding the cost of full Byzantine agreement in the common case, by using signed and dated certificates to cache the authorization granted through Byzantine operations.

Both Farsite's intended workload and its expected machine characteristics are those typically observed on desktop machines in academic and corporate settings. These workloads exhibit high access locality, a low persistent update rate, and a pattern of read/write sharing that is usually sequential and rarely concurrent [22, 48]. The expected machine characteristics include a high fail-stop rate (often just a user turning a machine off for a while) [6] and a low but significant rate [41] of malicious or opportunistic subversion. In our design, analysis, evaluation, and discussion, we focus on this environment, but we note that corporate administrators might choose to supplement Farsite's reliability and security by adding userless machines to the system or even running entirely on machines in locked rooms.

Farsite requires no central administration beyond that needed to initially configure a minimal system and to authenticate new users and machines as they join the system. Administration is mainly an issue of signing certificates: Machine certificates bind machines to their public keys; user certificates bind users to their public keys; and namespace certificates bind namespace roots to their managing machines. Beyond initially signing the namespace certificate and subsequently signing certificates for new machines and users, no effort is required from a central administrator.

There are many directions we could have explored in the Farsite design space that we have chosen not to. Farsite is not a high-speed parallel I/O system such as SGI's XFS [43], and it does not efficiently support large-scale write sharing of files. Farsite is intended to emulate the behavior of a traditional local file system, in particular NTFS [42]; therefore, it introduces no new user-visible semantics, such as an object-model interface, transactional support, versioning [39], user-specifiable file importance, or Coda-like [22] hooks for application-specific conflict resolvers to support concurrent file updates during disconnected operation.

We have implemented most – but not all – of the design described in this paper. The exceptions, which mainly relate to scalability and crash recovery, are itemized in section 6 and identified throughout the text with the term *Farsite design*, indicating a mechanism that we have designed but not yet implemented.

The following section presents a detailed overview of the system. Section 3, the bulk of the paper, describes the mechanisms that provide Farsite's key features. Section 4 describes our prototype implementation. Section 5 presents analytical arguments that show the Farsite design to be scalable, and it presents empirical measurements that show our prototype's performance to be acceptable. We discuss future work in section 6, related work in section 7, and conclusions in section 8.

2. System Overview

This section begins by presenting the assumptions and technology trends that underlie Farsite's design. Then, it introduces the design by explaining the fundamental concept of namespace roots, describing our trust model and certification mechanisms, outlining the system architecture, and describing a few necessary semantic differences between Farsite and a local file system.

2.1 Design Assumptions

Most of our design assumptions stem from the fact that Farsite is intended to run on the desktop workstations of a large corporation or university. Thus, we assume a maximum scale of ~10^5 machines, none of which are dedicated servers, and all of which are interconnected by a high-bandwidth, low-latency network whose topology can be ignored. Machine availability is assumed to be lower than that of dedicated servers but higher than that of hosts on the Internet; specifically, we expect the majority of machines to be up and accessible for the majority of the time. We assume machine downtimes to be generally uncorrelated and permanent machine failures to be mostly independent. Although Farsite tolerates large-scale read-only sharing and small-scale read/write sharing, we assume that no files are both read by many users and also frequently updated by at least one user. Empirical data [6, 48] corroborates this set of assumptions.

We assume that a small but significant fraction of users will maliciously attempt to destroy or corrupt file data or metadata, a reasonable assumption for our target environment but an unreasonably optimistic one on the open Internet. We assume that a large fraction of users may independently and opportunistically attempt to read file data or metadata to which they have not been granted access. Each machine is assumed to be under the control of its immediate user, and a party of malicious users can subvert only machines owned by members of the party.

We assume that the local machine of each user can be trusted to perform correctly for that user, and no user-sensitive data persists beyond user logoff or system reboot. This latter assumption is known to be false without a technique such as crypto-paging [50], which is not employed by any commodity operating system including Windows, the platform for our prototype.

2.2 Enabling Technology Trends

Two technology trends are fundamental in rendering Farsite's design practical: a general increase in unused disk capacity and a decrease in the computational cost of cryptographic operations relative to I/O.

A very large fraction of the disk space on desktop workstations is unused, and disk capacity is increasing at a faster rate than disk usage. In 1998, measurements of 4800 desktop workstations at Microsoft [13] showed that 49% of disk space was unused. In 1999 and 2000, we measured the unused portions to be 50% and 58%.

As computation speed has increased with Moore's Law, the cost of cryptographic operations has fallen relative to the cost of the I/O operations that are the mainstay of a file system. We measured a modern workstation and found a symmetric encryption bandwidth of 72 MB/s and a one-way hashing bandwidth of 53 MB/s, both of which exceed the disk's sequential-I/O bandwidth of 32 MB/s. Encrypting or decrypting 32 KB of data adds roughly one millisecond of latency to the file-read/write path. Computing an RSA signature with a 1024-bit key takes 6.5 milliseconds, which is less than one rotation time for a 7200-RPM disk.

The large amount of unused disk capacity enables the use of replication for reliability, and the relatively low cost of strong cryptography enables distributed security.

2.3 Namespace Roots

The primary construct established by a file system is a hierarchical directory namespace, which is the logical repository for files. Since a namespace hierarchy is a tree, it has to have a root. Rather than mandating a single namespace root for any given collection of machines that form a Farsite system, we have chosen to allow the flexibility of multiple roots, each of which can be regarded as the name of a virtual file server that is collaboratively created by the participating machines. An administrator creates a namespace root simply by specifying a unique (for that administrator) root name and designating a set of machines to manage the root. These machines will form a Byzantine-fault-tolerant group (see subsection 2.5), so it is not crucial to select a specially protected set of machines to manage the root.

2.4 Trust and Certification

The security of any distributed system is essentially an issue of managing trust. Users need to trust the authority of machines that offer to present data and metadata. Machines need to trust the validity of requests from remote users to modify file contents or regions of the namespace. Security components that rely on redundancy need to trust that an apparently distinct set of machines is truly distinct and not a single malicious machine pretending to be many, a potentially devastating attack known as a Sybil attack [11].

Farsite manages trust using public-key-cryptographic certificates. A certificate is a semantically meaningful data structure that has been signed using a private key. The principal types of certificates are namespace certificates, user certificates, and machine certificates. A *namespace certificate* associates the root of a file-system namespace with a set of machines that manage the root metadata. A *user certificate* associates a user with his personal public key, so that the user identity can be validated for access control. A *machine certificate* associates a machine with its own public key, which is used for establishing the validity of the machine as a physically unique resource.

Trust is bootstrapped by fiat: Machines are instructed to accept the authorization of any certificate that can be validated with one or more particular public keys. The corresponding private keys are known as *certification authorities* (*CAs*). Consider a company with a central certification department, which generates a public/private key pair to function as a CA. Every employee generates a public/private key pair, and the CA signs a certificate associating the employee's username with the public key. For a small set of machines, the CA generates public/private key pairs and embeds the public keys in a namespace certificate that designates these machines as managers of its namespace root. The CA then performs the following four operations for every machine in the company: (1) Have the machine generate a public/private key pair; (2) sign a certificate associating the machine with the public key; (3) install the machine certificate and root namespace certificate on the machine; and (4) instruct the machine to accept any certificate signed by the CA.

Farsite's certification model is more general than the above paragraph suggests, because machine certificates are not signed directly by CAs but rather are signed by users whose certificates designate them as authorized to certify machines, akin to Windows' model for domain administration. This generalization allows a company to separate the responsibilities of authenticating users and machines, e.g., among HR and IT departments. A single machine can participate in multiple CA domains, but responsibility granted by one CA is only delegated to other machines authorized by that same CA.

A machine's private key is stored on the machine itself. In the Farsite design, each user private key is encrypted with a symmetric key derived from the user's password and then stored in a globally-readable directory in Farsite, where it can be accessed upon login. CA private keys should be kept offline, because the entire security of a Farsite installation rests on their secrecy.

User or machine keys can be revoked by the issuing CA, using the standard techniques [29]: Signed revocation lists are periodically posted in a prominent, highly replicated location. All certificates include an expiration date, which allows revocation lists to be garbage collected.

2.5 System Architecture

We now present a high-level overview of the Farsite architecture, beginning with a simplified system and subsequently introducing some of Farsite's key aspects.

2.5.1 Basic System

Every machine in Farsite may perform three roles: It is a *client*, a member of a *directory group*, and a *file host*, but we initially ignore the latter of these. A client is a machine that directly interacts with a user. A directory group is a set of machines that collectively manage file information using a Byzantine-fault-tolerant protocol [8]: Every member of the group stores a replica of the information, and as the group receives client requests, each member processes these requests deterministically, updates its replica, and sends replies to the client. The Byzantine protocol guarantees data consistency as long as fewer than a third of the machines misbehave.

Consider a system that includes several clients and one directory group. For the moment, imagine that the directory group manages all of the file-system data and metadata, storing it redundantly on all machines in the group. When a client wishes to read a file, it sends a message to the directory group, which replies with the contents of the requested file. If the client updates the file, it sends the update to the directory group. If another client tries to open the file while the first client has it open, the directory group evaluates the specified sharing semantics requested by the two clients to determine whether to grant the second client access.

2.5.2 System Enhancements

The above-described system provides reliability and data integrity through Byzantine replication. However, it may have performance problems since all file-system requests involve remote Byzantine operations; it does not provide data privacy from users who have physical access to the directory group members; it does not have the means to enforce user access control; it consumes a large amount of storage space since files are replicated on every directory group member, and it does not scale.

Farsite enhances the basic system in several ways. It adds local caching of file content on the client to improve read performance. Farsite's directory groups issue leases on files to clients, granting them access to the files for a specified period of time, so a client with an active lease and a cached copy of a file can perform operations entirely locally. Farsite delays pushing updates to the directory group, because most file writes are deleted or overwritten shortly after they occur [4, 48]. To protect user privacy and provide read-access control, clients encrypt written file data with the public keys of all authorized readers; and the directory group enforces write-access control by cryptographically validating requests from users before accepting updates.

Since Byzantine groups fail to make progress if a third or more of their members fail, directory groups require a high replication factor (e.g., 7 or 10) [8]. However, since the vast majority of file-system data is opaque file content, we can offload the storage of this content onto a smaller number of other machines ("file hosts"), using a level of indirection. By keeping a cryptographically secure hash of the content on the directory group, putative file content returned from a file host can be validated by the directory group, thereby preventing the file host from corrupting the data. Therefore, Farsite can tolerate the failure of all but one machine in a set of file hosts, thus allowing a far lower replication factor (e.g., 3 or 4) [6] for the lion's share of file-system data.

As machines and users join a Farsite system, the volume of file-system metadata will grow. At some point, the storage and/or operation load will overwhelm the directory group that manages the namespace. The Farsite design addresses this problem by allowing the directory group to delegate a portion of its namespace to another directory group. Specifically, the group randomly selects, from all of the machines it knows about, a set of machines that it then instructs to form a new directory group. The first group collectively signs a new namespace certificate delegating authority over a portion of its namespace to the newly formed group. This group can further delegate a portion of its region of the namespace. Because each file-system namespace forms a hierarchical tree, each directory group can delegate any sub-subtree of the subtree that it manages without involving any other extant directory groups.

In this enhanced system, when a client wishes to read a file, it sends a message to the directory group that manages the file's metadata. This group can prove to the client that it has authority over this directory by presenting a namespace certificate signed by its parent group, another certificate signed by its parent's parent, and so on up to the root namespace certificate signed by a CA that the client regards as authoritative. The group replies with these namespace certificates, a lease on the file, a one-way hash of the file contents, and a list of file hosts that store encrypted replicas of the file. The client retrieves a replica from a file host and validates its contents using the one-way hash. If the user on the client machine has read access to the file, then he can use his private key to decrypt the file. If the client updates the file, then, after a delay, it sends an updated hash to the directory group. The directory group verifies the user's permission to write to the file, and it instructs the file hosts to retrieve copies of the new data from the client. If another client tries to open the file while the first client has an active lease, the directory group may need to contact the first client to see whether the file is still in use, and the group may recall the lease to satisfy the new request.

2.6 Semantic Differences from NTFS

Despite our goal of emulating a local NTFS file system as closely as possible, performance or behavioral issues have occasionally motivated semantics in Farsite that diverge from those of NTFS.

If many (e.g., more than a thousand) clients hold a file open concurrently, the load of consistency management – via querying and possibly recalling leases – becomes excessive, reducing system performance. To prevent this, the Farsite design places a hard limit on the number of clients that can have a file open for concurrent writing and a soft limit on the number that can have a file open for concurrent reading. Additional attempts to open the file for writing will fail with a sharing violation. Additional attempts to open the file for reading will not receive a handle to the file; instead, they will receive a handle to a snapshot of the file, taken at the time of the open request. Because it is only a snapshot, it will not change to reflect updates by remote writers, and so it may become stale. Also, an open snapshot handle will not prevent another client from opening the file for exclusive access, as a real file handle would. An application can query the Farsite client to find out whether it has a snapshot handle or a true file handle, but this is not part of NTFS semantics.

NTFS does not allow a directory to be renamed if there is an open handle on a file in the directory or in any of its descendents. In a system of 10^5 machines, there will almost always be an open handle on a file somewhere in the namespace, so these semantics would effectively prevent a directory near the root of the tree from ever being renamed. Thus, we instead implement the Unix-like semantics of not name-locking an open file's path.

The results of directory rename operations are not propagated synchronously to all descendent directory groups during the rename operation, because this would unacceptably retard the rename operation, particularly for directories near the root of the namespace tree. Instead, they are propagated lazily, so they might not be immediately visible to all clients.

Windows applications can register to be informed about changes that occur in directories or directory subtrees. This notification is specified to be best-effort, so we support it without issuing read leases to registrants. However, for reasons of scalability, we only support notification on single directories and not on subtrees.

3. File System Features

This section describes the mechanisms behind Farsite's key features, which include reliability and availability, security, durability, consistency, scalability, efficiency, and manageability.

3.1 Reliability and Availability

Farsite achieves reliability (long-term data persistence) and availability (immediate accessibility of file data when requested) mainly through replication. Directory metadata is replicated among members of a directory group, and file data is replicated on multiple file hosts. Directory groups employ Byzantine-fault-tolerant replication, and file hosts employ raw replication.

For redundantly storing file data, Farsite could have used an erasure coding scheme [3] rather than raw replication. We chose the latter in part because it is simpler and in part because we had concerns about the additional latency introduced by fragment reassembly of erasure-coded data during file reads. Some empirical data [25] suggests that our performance concerns might have been overly pessimistic, so we may revisit this decision in the future. The file replication subsystem is a readily separable component of both the architecture and the implementation, so it will be straightforward to replace if we so decide.

With regard to reliability, replication guards against the permanent death of individual machines, including data-loss failures (such as head crashes) and explicit user decommissioning. With regard to availability, replication guards against the transient inaccessibility of individual machines, including system crashes, network partitions, and explicit shutdowns. In a directory group of R_D members, metadata is preserved and accessible if no more than $\lfloor (R_D - 1) / 3 \rfloor$ of the machines die. For files replicated on R_F file hosts, file data is preserved and accessible if at least one file host remains alive.

In the Farsite design, when a machine is unavailable for an extended period of time, its functions migrate to one or more other machines, using the other replicas of the file data and directory metadata to regenerate that data and metadata on the replacement machines. Thus, data is lost permanently only if too many machines fail within too small a time window to permit regeneration.

Because the volume of directory data is much smaller than that of file data, directory migration is performed more aggressively than file-host migration: Whenever a directory group member is down or inaccessible for even a short time, the other members of the group select a replacement randomly from the set of accessible machines they know about. Since low-availability machines are – by definition – up for a smaller fraction of time than high-availability machines, they are more likely to have their state migrated to another machine and less likely to be an accessible target for migration from another machine. These factors bias directory-group membership toward highly available machines, without introducing security-impairing non-randomness into member selection. This bias is desirable since

Byzantine agreement is only possible when more than two thirds of the replicas are operational. The increase in the (light) metadata workload on high-availability machines is compensated by a small decrease in their (heavy) file storage and replication workload.

Farsite improves global file availability by continuously relocating file replicas at a sustainable background rate [14]. Overall mean file uptime is maximized by achieving an equitable distribution of file availability, because low-availability files degrade mean file uptime more than high-availability files improve it. Therefore, Farsite successively swaps the machine locations of replicas of high-availability files with those of low-availability files, which progressively equalizes file availability. Simulation experiments [14] driven by actual measurements of desktop machine availability show that Farsite needs to swap 1% of file replicas per day to compensate for changes in machine availability.

File availability is further improved by caching file data on client disks. These caches are not fixed in size but rather hold file content for a specified interval called the *cache retention period* (roughly one week) [6].

3.2 Security

3.2.1 Access Control

Farsite uses different mechanisms to enforce write- and read-access controls. Because directory groups only modify their shared state via a Byzantine-fault-tolerant protocol, we trust the group not to make an incorrect update to directory metadata. This metadata includes an *access control list* (*ACL*) of public keys of all users who are authorized writers to that directory and to files therein. When a client establishes cryptographically authenticated channels to a directory group's members, the channel-establishment protocol involves a user's private key, thereby authenticating the messages on that channel as originating from a specific user. The directory group validates the authorization of a user's update request before accepting the update.

Because a single compromised directory-group member can inappropriately disclose information, Farsite enforces read-access control via strong cryptography, as described in the next subsection.

3.2.2 Privacy

Both file content and user-sensitive metadata (meaning file and directory names) are encrypted for privacy.

When a client creates a new file, it randomly generates a symmetric *file key* with which it encrypts the file. It then encrypts the file key using the public keys of all authorized readers of the file, and it stores the file key encryptions with the file, so a user with a corresponding private key can decrypt the file key and therewith the file. Actually, there is one more level of indirection because of the need to identify and coalesce identical files (see subsection 3.6.1) even if they are encrypted with different user keys: The client first computes a one-way hash of each block of the file, and this hash is used as a key for encrypting the block. The file key is used to encrypt the hashes rather than to encrypt the file blocks directly. We call this technique *convergent encryption* [12], because identical file plaintext converges to identical ciphertext, irrespective of the user keys. Performing encryption on a block level enables a client to write an individual block without having to rewrite the entire file. It also enables the client to read individual blocks without having to wait for the download of an entire file from a file host.

To prevent members of a directory group from viewing file or directory names, they are encrypted by clients before being sent to the group, using a symmetric key that is encrypted with the public keys of authorized directory readers and stored in the directory metadata [15]. To prevent a malicious client from encrypting a syntactically illegal name [31], the Farsite design uses a technique called *exclusive encryption*, which augments a cryptosystem in a way that guarantees that decryption can only produce legal names no matter what bit string is given as putative ciphertext [15].

3.2.3 Integrity

As long as fewer than one third of the members of a directory group are faulty or malicious, the integrity of directory metadata is maintained by the Byzantine-fault-tolerant protocol. Integrity of file data is ensured by computing a Merkle hash tree [30] over the file data blocks, storing a copy of the tree with the file, and keeping a copy of the root hash in the directory group that manages the file's metadata. Because of the tree, the cost of an in-place file-block update is logarithmic – rather than linear – in the file size. The hash tree also enables a client to validate any file block in logarithmic time, without waiting to download the entire file. The time to validate the entire file is linear in the file size, not log-linear, because the count of internal hash nodes is proportional to the count of leaf content nodes.

3.3 Durability

When an application creates, modifies, renames, or deletes a file or directory, these updates to metadata are committed only on the client's local disk and not by a Byzantine operation to the directory group, due to the high cost of the latter. The updates are written into a log (much as in Coda [22]), which is compressed when possible via techniques such as removing matching create-delete operation pairs. The log is pushed back to the directory group periodically and also when a lease is recalled, as described in subsection 3.4. Because client machines are not fully trusted, the directory group verifies the legality of each log entry before performing the update operation that it specifies.

When a client machine reboots after a crash, it needs to send committed updates to the directory group and have the group accept them as validly originating from the user. The two obvious ways of accomplishing this are unacceptable: Private-key-signing every committed update would be prohibitively expensive (roughly a disk seek time), and holding the user's private key on the client through a crash would open a security hole.

Instead, when first contacting a directory group, a client generates a random *authenticator key* and splits it into secret shares [46], which it distributes among members of the directory group. This key is not stored on the client's local disk, so it is unavailable to an attacker after a crash (modulo the lack of crypto-paging in the underlying operating system – see subsection 2.1). With this key, the client signs each committed update using a message authentication code (MAC) [29]. (Symmetric-key MACs are much faster than public-key signatures.) When recovering from a crash, the client sends the MAC to the directory group along with the locally committed updates. In a single transaction, the group members first batch the set of updates, then jointly reconstruct the authenticator key, validate the batch of updates, and discard the key. Once the key has been reconstructed, no further updates will be accepted. (The recovery phase of this process has not yet been implemented.)

Modifying a file (unlike creating, renaming, or deleting it) affects not only the file metadata but also the file content. It is necessary to update the content atomically with the metadata; otherwise, they may be left in an inconsistent state following a crash, and the file content will be unverifiable. For the rare [48] cases when an existing block is overwritten, the new content is logged along with the metadata before the file is updated, so a partial write can be rolled forward from the log. When a client appends a new block to the end of a file, the content need not be logged; it is sufficient to atomically update the file length and content hash after writing the new block. If a crash leaves a partially written block, it can be rolled backward to an empty block [20].

3.4 Consistency

The ultimate responsibility for consistency of file data and directory metadata lies with the directory group that manages the file or directory. However, temporary, post-hoc-verifiable control can be loaned to client machines via a lease mechanism. There are four *classes* of leases in Farsite: content leases, name leases, mode leases, and access leases. They are described in the following four subsections.

3.4.1 Data Consistency

Content leases govern which client machines currently have control of a file's content. There are two content-lease *types*: *read/write* and *read-only*, and they support single-writer multi-reader semantics. A read-only lease assures a client that the file data it sees is not stale. A read/write lease entitles a client to perform write operations to its local copy of the file. Applications are not aware of whether their clients hold content leases.

When an application opens a file, the client requests a content lease from the directory group that manages the file. When the application closes the file, the client does not immediately cancel its lease, since it may soon need to open that file again [48]. If another client makes a valid request for a new content lease while the first client holds a conflicting lease, the directory group will recall the lease from the first client. When this client returns the lease, it will also push all of its logged updates to the directory group. The group will apply these updates before issuing the new content lease to the other client, thereby maintaining consistency.

Since read/write and write/write file sharing on workstations is usually sequential [22, 48], it is often tolerably efficient to ping-pong leases between clients as they make alternate file accesses. However, as a performance optimization, the Farsite design includes a mechanism similar to that in Sprite [35]: redirecting concurrent non-read-only accesses through a single client machine. This approach is not scalable, but Farsite is not designed for large-scale write sharing. We have not yet determined an appropriate policy for when concurrent non-read-only accesses should switch from lease ping-pong to single-client serialization.

Content leases have variable granularity: A lease may cover a single file, or it may cover an entire directory of files, similar to a volume lease in AFS [21]. Directory groups may issue broader leases than those requested by the client, if no other clients have recently made conflicting accesses to the broader set of files.

Because clients can fail or become disconnected, they may be unable to respond to a lease recall. To prevent this situation from rendering a file permanently inaccessible, leases include expiration times. The lease time varies depending upon the type of lease (i.e., read-only leases last for longer than read/write leases) and upon the observed degree of sharing on the file. The directory group treats lease expiration identically to a client's closing the file without making further updates.

There are several options for handling lease expiration on a disconnected client: The most pessimistic strategy is to close all handles to the file and drop any logged updates, since it may not be possible to apply the updates if the file is modified by another client after the lease expires. We could, however, save the logged updates and, if the file has not changed, apply them after re-establishing communications with the directory group. We may wish to keep file handles open for read access to potentially stale data, or – most optimistically – even allow further updates, since we might be able to apply these later. We have not yet explored this space nor implemented any mechanisms.

Since the Windows directory-listing functions are specified to be best-effort, we support them using a snapshot of directory contents rather than with a lease.

As described in subsection 2.7, the number of leases issued per file is limited for performance reasons.

3.4.2 Namespace Consistency

Shared file-system namespaces commonly [34] contain regions that are private to particular users. To allow clients to modify such regions without having to frequently contact the directory group that manages the region, Farsite introduces the concept of *name leases*.

Name leases govern which client machine currently has control over a name in the directory namespace. There is only one type of name lease, but it has two different meanings depending on whether a directory (or file) with that name exists: If there is no such extant name, then a name lease entitles a client to create a file or directory with that name. If a directory with the name exists, then the name lease entitles a client to create files or subdirectories under that directory with any non-extant name. This dual meaning implies that when a client uses a name lease to create a new directory, it can then immediately create files and subdirectories in that directory. To rename a file or directory, a client must obtain a name lease on the target name.

Like content leases, name leases can be recalled if a client wants to create a name that falls within the scope of a name lease that has been issued to another client. The discussion above regarding expiration of content leases also applies to name leases.

3.4.3 Windows File-Sharing Semantics

Windows supports application-level consistency by providing explicit control over file-sharing semantics. When an application opens a file, it specifies two parameters: (1) the access mode, which is the types of access it wants, and (2) the sharing mode, which is the types of access it is willing to concurrently allow others. There are many different access modes, but from the perspective of a distributed system, they can be distilled down to three (and finer distinctions can be enforced locally by the client): read access, write access, and delete access. There are also three sharing modes: read sharing, write sharing, and delete sharing, which permit other applications to open the file for read access, write access, and delete access, respectively.

As an example, if an application tries to open a file with read access mode, the open will fail if another application has the file open without read sharing mode. Conversely, if an application tries to open a file without read sharing mode, the open will fail if another application has the file open with read access mode.

To support these semantics, Farsite employs six types of *mode leases*: *read*, *write*, *delete*, *exclude-read*, *exclude-write*, and *exclude-delete*. When a client opens a file, Farsite translates the requested access and share modes into the corresponding mode-lease types: Each access mode implies a need for the corresponding mode lease, and the lack of each sharing mode implies a need for the corresponding exclude lease. For example, if an application opens a file for read access, the client will request a read mode lease, and if the open allows only read sharing, then it will also request exclude-write and exclude-delete mode leases. When a directory group processes a client's open request, it determines whether it can issue the requested leases without conflicting with any extant mode leases on the file. If it cannot, it contacts the client or clients who hold conflicting mode leases to see if they are willing to have their leases revoked or downgraded, which they may be if their applications no longer have open handles on the file. The mode-lease conflicts are the obvious ones: read conflicts with exclude-read, write with exclude-write, and delete with exclude-delete.

3.4.4 Windows Deletion Semantics

Windows has surprisingly complex deletion semantics: A file is deleted by opening it, marking it for deletion, and closing it. Since there may be multiple handles open on a file, the file is not truly deleted until the last handle on a deletion-marked file is closed. While the file is marked for deletion, no new handles may be opened on the file, but any application that has an open delete-access handle can clear the deletion mark, thereby canceling the deletion and also permitting new handles on the file to be opened.

To support these semantics, the Farsite design employs three types of *access leases*: *public*, *protected*, and *private*. A public access lease indicates that the lease holder has the file open. A protected lease includes the meaning of a public lease, and it further indicates that no other client will be granted access without first contacting the lease holder. A private lease includes the meaning of a protected lease, and it further indicates that no other client has any access lease on the file. When a client opens a file, the managing directory group checks to see whether it has issued a private or protected access lease to any other client. If it has, it downgrades this lease to a public lease, thereby forcing a metadata pushback (as described in subsection 3.3), before issuing an access lease to the new client.

To mark a file for deletion, a client must first obtain a private or protected access lease on the file. If the directory group thereafter receives a different client's request to open the file, downgrading this access lease to a public lease will force a metadata pushback, thus informing the directory group of whether the client has delete-marked the file. If the file has been marked, then the directory group will deny the open request.

If a client has a private access lease on a file, it is guaranteed that no other client has an open handle, so it can delete the file as an entirely local operation.

3.5 Scalability

The Farsite design uses two main mechanisms to keep a node's computation, communication, and storage from growing with the system size: hint-based pathname translation and delayed directory-change notification.

When a directory group becomes overloaded, at least $\lceil (2 R_D + 1) / 3 \rceil$ of its R_D members can sign a certificate that delegates part of its namespace to another group. When a client attempts to open a file or directory with a particular pathname, it needs to determine which group of machines is responsible for that name. The basis mechanism is to contact successive directory groups until the responsible group is found. This search begins with the group that manages the root of the namespace, and each contacted group provides a delegation certificate that indicates which group to contact next.

This basis approach clearly does not scale, because all pathname translations require contacting the root directory group. Therefore, to perform translations in a scalable fashion, the Farsite design uses a hint-based scheme that tolerates corrupt directory group members, group membership changes, and stale delegations: Each client maintains a cache of pathnames and their mappings to directory groups, similar to prefix tables in Sprite [35]. A client translates a path by finding the longest-matching prefix in its cache and contacting the indicated directory group. There are three cases: (1) Because of access locality, the most common case is that the contacted group manages the pathname. (2) If the group manages only a path prefix of the name, then it replies with all of its delegation certificates, which the client adds to its hint cache. (3) If the group does not manage a path prefix of the name, then it informs the client, which removes the stale hint that led it to the incorrect group. In the latter two cases, the client again finds the longest-matching prefix and repeats the above steps. Because of the signed delegation certificates, no part of this protocol requires Byzantine operations. Each step either removes old information or adds new information about at least one pathname component, so the translation terminates after no more than twice the number of components in the path being translated.

Directory-change notification is a Windows mechanism that allows applications to register for callbacks when changes occur to a specified directory. The archetypal example is Windows Explorer, which registers a notification for the directory currently displayed. Since Windows specifies this notification to be best-effort, the Farsite design supports it in a delayed manner: For any directory for which a notification has been registered, the directory group packages the complete directory information, signs it to authenticate its contents, and sends it to the registered clients. The transmission work can be divided among members of the directory group and also among clients using application-level multicast [16]. Farsite clients automatically register for directory-change notification when a user lists a directory, so repeat listings need not make multiple remote requests.

3.6 Efficiency

3.6.1 Space Efficiency

File and directory replication dramatically increase the storage requirements of the system. To make room for this additional storage, Farsite reclaims space from incidentally duplicated files, such as workgroup-shared documents or multiple copies of common applications.

Measurements of 550 desktop file systems at Microsoft [6] show that almost half of all occupied disk space can be reclaimed by eliminating duplicates. The Farsite design detects files with duplicate content by storing file-content hashes in a scalable, distributed, fault-tolerant database [12]. It then co-locates replicas of identical files onto the same set of file hosts, where they are coalesced by Windows' Single Instance Store [7].

3.6.2 Time Efficiency (Performance)

Many of the mechanisms already described have been designed in part for their effect on system performance. Caching encrypted file content on client disks improves not only file availability but also file-read performance (further improved by caching decrypted file content in memory). Farsite's various lease mechanisms and its hint-based pathname translation not only reduce the load on directory groups but also eliminate the latency of network round trips. Performance is the primary motivation behind limiting the count of leases per file, using Merkle trees for data integrity, and MAC-logging metadata updates on clients.

In addition, Farsite inserts a delay between the creation or update of a file and the replication of the new file content, thus providing an opportunity to abort the replication if the operation that motivated it is superseded. Since the majority of file creations and updates are followed by deletions or other updates shortly thereafter [4, 48], this delay permits a dramatic reduction in network file-replication traffic [6].

3.7 Manageability

3.7.1 Local-Machine Administration

Although Farsite requires little central administration, users may occasionally need or want to perform local-system administrative tasks, such as hardware upgrades, software upgrades, and backup of private data.

When using a local file system, upgrading a machine's hardware by replacing the disk (or by replacing the entire machine) necessitates copying all file-system structure and content from the old disk to the new one. In Farsite, file data and metadata are replicated on other machines for reliability, so removing the old disk (or decommissioning an old machine) is merely a special case of hardware failure. In its capacities as file host and directory group member, the machine's functions will be migrated; and in its capacity as a client, the hint cache and content cache will gradually refill. However, since users replace their machines far more frequently than machines fail, Farsite's reliability is substantially improved by providing users a means to indicate their intention to decommission a machine or disk, spurring a preemptive migration of the machine's functions [6].

To address the need for upgrades and bug fixes, Farsite supports interoperation between machines running different versions of its software by including major and minor version numbers in connection-establishment messages. Minor-version changes can – by definition – be ignored, and later versions of the software can send and understand all earlier major versions of messages. This backward compatibility permits users to self-pace their upgrades, using a suggestion-based model similar to Windows Update [32]. Updated executables need not be sourced centrally: If they are signed using a private key whose public counterpart is baked into the Farsite code, upgrades can be obtained from any other machine running a more recent version of Farsite code.

Backup processes are commonly used for two purposes: reliability and archiving. In Farsite, there is little need for the former, since the multiple on-line copies of each file on independent machines should be more reliable than a single extra copy on a backup tape whose quality is rarely verified. For archival purposes, automatic on-line versioning [40] would be a more valuable system addition than manual off-line backup. However, if users still wish to backup their own regions of the namespace, existing backup utilities should work fine, except for two problems: pollution of the local cache and weakening of privacy from storing decrypted data on tape. To address the first problem, we could exploit a flag that Windows backup utilities use to indicate their purpose in opening files: Farsite could respond to this flag by not locally caching the file. The second problem, which is outside of Farsite's domain, could be addressed by an encrypting backup utility.

3.7.2 Autonomic System Operation

Farsite administers itself in a distributed, Byzantine-fault-tolerant fashion. Self-administration tasks are either lazy follow-ups scheduled after client-initiated operations or continual background tasks performed periodically. The details of these tasks (file replication, replica relocation, directory migration, namespace delegation, and duplicate-file identification/coalescing) have been described above, but in this section we describe their substrate: two semantic extensions to the conventional model of Byzantine fault-tolerance, *timed Byzantine operations* and *Byzantine outcalls*.

In the standard conception of Byzantine fault-tolerant distributed systems [8], an operation is initiated by a single machine and performed by a Byzantine-fault-tolerant replica group. This modifies the shared state of the group members and returns a result to the initiator.

For lazy and periodic tasks (e.g., replica relocation), directory groups need to initiate operations in response to a timer, rather than in response to a client request. This is complicated because Byzantine replicas must perform operations in lock step, but the clocks of group members cannot be perfectly synchronized. Farsite supports timed Byzantine operations via the following mechanism: The Byzantine-replicated state includes R_D *member time* values, each associated with one of the R_D machines in the group. The kth largest member time (where $k = \lfloor (R_D - 1) / 3 + 1 \rfloor$) is regarded as the *group time*. When a machine's local time indicates that a timed operation should be performed, it invokes the Byzantine protocol to update its replicated member time to the machine's local time. By modifying one of the member times, this update may change the group time, in which case the group performs all operations scheduled to occur at or before the new group time.

Self-administration tasks invert the standard model of Byzantine-fault-tolerance: The directory group invokes an operation (e.g., instruction to create a file replica) on a single remote machine, which sends a reply back to the group. We perform these Byzantine outcalls using a hint-based scheme: A Byzantine operation updates the replicated state to enqueue a request to a remote machine. Then, members of the directory group, acting as individuals, send hint messages to the remote machine, suggesting that it invoke a Byzantine operation to see if the directory group has any work for it to do. When the machine invokes the Byzantine operation, the request is dequeued and returned to the invoking machine. The hint messages are staggered in time, so in the common case, after one member sends a hint message, the remote machine dequeues its request, preventing the other members from sending redundant hints. If the machine needs to reply to the group, it does so by invoking another Byzantine operation.

4. Implementation

Farsite is implemented as two components: a user-level service (daemon) and a kernel-level driver. The driver implements only those operations whose functionality or performance necessitates placement in the kernel: exporting a file-system interface, interacting with the cache manager, and encrypting and decrypting file content. All other functions are implemented in user level: managing the local file cache, fetching files from remote machines, validating file content, replicating and relocating files, issuing leases, managing metadata, and executing the Byzantine-fault-tolerant protocol.

Windows' native remote file system is RDR / SRV, a pair of file-system drivers that communicate via the CIFS protocol [42]. RDR lives under RDBSS, a driver that provides a general framework for implementing network file systems. RDR accepts file-system calls from applications and redirects them via the network to SRV running on another machine, which communicates with NTFS to satisfy the request.

We took advantage of this framework by implementing Farsite's kernel driver under RDBSS. The *Farsite Mini Redirector* (*FMR*) accepts file-system calls, interacts with Farsite's user-level component, uses NTFS as a local persistent store, and performs encryption and decryption on the file-I/O read and write paths.

Our user-level component is called the *Farsite Local Manager* (*FLM*). The control channel by which FLMs on different machines communicate is an encrypted, authenticated connection established on top of TCP. The data channel by which encrypted file content is retrieved from other machines is a CIFS connection established by the RDR and SRV components of the two machines. This architecture is illustrated in Fig. 1.

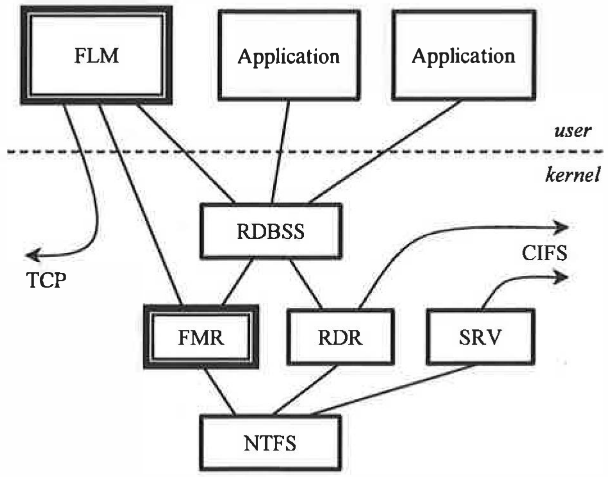


Fig. 1: Farsite components in system context

5. Evaluation

In the following two subsections, we evaluate Farsite's scalability by calculating the expected central loads of certification, directory access, and path translation as a function of system size; and we evaluate Farsite's performance by benchmarking our prototype.

5.1 Scalability Analysis

Farsite's scalability target is ~10^5 machines. Scale is potentially limited by two points of centralization: certification authorities and root directory groups.

Farsite's certification authorities sign certificates offline rather than interactively processing verifications online. Therefore, their workload is determined by the number of private-key signatures they have to perform, not by the much greater number of verifications required. If a company issues machine and user certificates with a one-month lifetime, then in a system of 10^5 machines and users, the CA will have to sign 2×10^5 certificates per month. Each RSA private-key signature takes less than 10 ms of CPU time, so the total computational workload is less than one CPU-hour per month.

Load on a directory group can be categorized into two general classes: the direct load of accesses and updates to the metadata it manages and the indirect load of performing pathname translations for directory groups that manage directories lower in the hierarchy. As a Farsite installation grows, the count of client machines in the system, all of which could concurrently access the same directory, also grows. Since the count of files per directory is independent of system scale [6], the direct load is bounded by limiting the number of outstanding leases per file (see subsection 2.7) and by distributing the transmission of directory-change notifications (see subsection 3.5) using application-level multicast, which is demonstrably efficient for such batch-update processes [16].

The indirect load on a directory group due to pathname translations is heavily reduced by the hint-caching mechanism described in subsection 3.5. When a new client joins the system, its first file access contacts the root directory group. If machines have a mean lifetime of one year [6], then a system of ~10^5 machines will see roughly 300 new machines per day, placing a trivial translation load on the root group from initial requests. Since the group responds with all of its delegation certificates, each client should never need to contact the root group again. Before a directory group's delegation certificates expire, it issues new certificates with later expiration dates and passes them down the directory-group hierarchy and on to clients.

5.2 Performance Measurements

For our performance evaluation, we configured a five-machine Farsite system using 1-GHz P3 machines, each with 512 MB of memory, two 114-GB Maxtor 4G120J6 drives, and a 100-Mbps Intel 82559 NIC, interconnected by a Cisco WS-C2948G network switch. Four machines served as file hosts and as members of a directory group, and one machine served as a client.

We collected a one-hour NTFS file-system trace from a developer's machine in our research group. The trace, collected from 11 AM to noon PDT on September 13, 2002, includes 450,164 file-system operations whose temporal frequency and type breakdown are closely representative of the file-system workload of this machine over the working hours in a measured three-week period. We replayed this trace on a Farsite client machine, and for comparison, we also replayed it onto a local NTFS partition and to a remote machine via CIFS.

Fig. 2 shows CDFs of operation timings during these experiments. For operations with very short durations (less than 7 μsec), Farsite is actually faster than NTFS, primarily due to a shorter code path. Nearly half of all operations are in this category. For the remainder, except for the slowest 1.4% of operations, Farsite's speed is between that of NTFS and CIFS.

We broke down the operation timings by operation type for the six types that took 99% of all I/O time. Farsite's mean operation durations are 2 to 4 times as long as those of NTFS for reads, writes, and closes; they are 9 times as long for opens; and they are 20 times as long for file-attribute and directory queries. Over the entire trace, Farsite displayed 5.5 times the file-I/O latency of NTFS. Some of this slowdown is due to making kernel/user crossings between the FMR and the FLM, and much of it is due to untuned code.

Relative to CIFS, Farsite's mean operation durations are 2 times as long on writes but only 0.4 times as long on reads and 0.7 times as long on queries. Overall, Farsite displayed 0.8 times the file-I/O latency of CIFS.

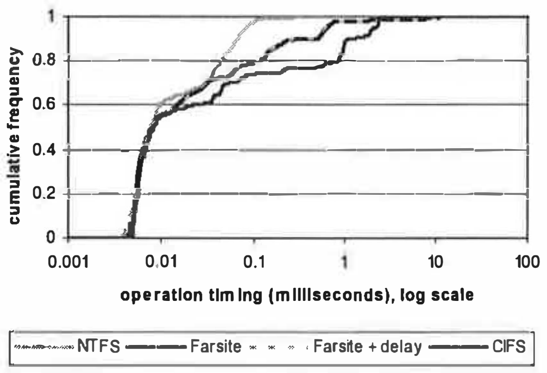


Fig. 2: CDF of trace-replay operation timings

Table 1: Andrew benchmark timings (seconds)

NTFS	CIFS	Farsite
1.9×10^4	3.4×10^4	3.7×10^4

To evaluate Farsite's sensitivity to network latency, we inserted a one-second delay into network transmissions. Because the vast majority of Farsite's operations are performed entirely locally, the effect on the log-scaled CDF was merely to stretch the thin upper tail to the right, which is nearly invisible in Fig 2. This delay did, however, add 10% to Farsite's total file-I/O latency.

Primarily because it is customary to do so, we also ran a version of the Andrew benchmark that we modified for Windows and scaled up by three orders of magnitude. It performs successive phases of creating directories, copying files, listing metadata, processing file content, and compiling the file-system code; each phase has a footprint of 5 – 11 GB. Table 1 shows total run times, but such Andrew benchmark results do not reflect a realistic workload. In particular, Farsite performs the directory-creation phase nearly twice as fast as NTFS, largely because NTFS is inefficient at creating a batch of millions of directories.

6. Future Work

Although we have implemented much of the Farsite design, several significant components remain, mainly those concerned with scalability (namespace delegation, distributed pathname translation, and directory-change notification) and those concerned with crash-recovery (directory-group membership change and torn-write repair). The mechanism for distributed duplicate detection is operational but not yet integrated into the rest of the system. Several smaller components have not yet been completed, including exclusive encryption of filenames, serializing multiple writers on a single machine, supporting lease expirations on clients, and full support for Windows deletion semantics.

Farsite requires two additional mechanisms for which we have not yet developed designs. First, to prevent a single user from consuming all available space in the system, we need a mechanism to enforce per-user space quotas; our intent is to limit each user's storage-space consumption to an amount proportional to that user's machines' storage-space contribution, with an expected proportionality constant near unity [6]. Second, Farsite relocates file replicas among machines according to the measured availability of those machines [14], which requires a mechanism to measure machine availability. Given Farsite's design assumptions, these mechanisms must be scalable, decentralized, fault-tolerant, and secure, making their design rather more demanding than one might initially presume.

7. Related Work

Farsite has many forebears in the history of network file systems. NFS [39] provides server-based, location-transparent file storage for diskless clients. AFS [21] improves performance and availability on disk-enabled workstations via leases and client-side file caching. In Sprite [35], clients use prefix tables for searching the file namespace. Coda [22] replicates files on multiple servers to improve availability. The xFS [1] file system decentralizes file-storage service among a set of trusted workstations, as does the Frangipani [45] file system running on top of the Petal [24] distributed virtual disk. All of these systems rely on trusted machines, and the decentralized systems (xFS and Frangipani) maintain per-client state that is proportional to the system size.

An important area of distributed-file-systems research, but one that is orthogonal to Farsite, is disconnected operation. The Coda [22], Ficus [37], and Bayou [44] systems researched this area extensively, and Farsite could adopt the established solution of application-specific resolvers for concurrent update conflicts.

Several earlier networked file systems have addressed one or more aspects of security. Blaze's Cryptographic File System [5] encrypts a single user's files on a client machine and stores the encrypted files on a server. BFS [8] replaces a single NFS server with a Byzantine-fault-tolerant replica group. SUNDR [27] guarantees file privacy, integrity, and consistency despite a potentially malicious server, but it does this by placing trust in all client machines (unlike Farsite, which requires only that each user trust the client machine he is directly using). SFS [26] constructs "self-certifying pathnames" by embedding hashes of public keys into file names, thus defending read-only data against compromised servers or compromised networks.

A number of distributed storage systems attempt to address the issue of scalability. Inspired by peer-to-peer file-sharing applications such as Napster [33], Gnutella [17], and Freenet [9], storage systems such as CFS [10] and PAST [38] employ scalable, distributed algorithms for routing and storing data. Widespread data distribution is employed by the Eternity Service's [2] replication system and by Archival Intermemory's [18] erasure-coding mechanism to prevent data loss despite attack by powerful adversaries, and PASIS [49] additionally employs secret sharing for data security. OceanStore [23] is designed to store all of the world's data (10^{23} bytes) using trans-continentally distributed, Byzantine-fault-tolerant replica groups to provide user-selectable consistency semantics. These systems have flat namespaces; they do not export file-system interfaces; and (with the exception of OceanStore) they are designed for publishing or archiving data, rather than for interactively using and updating data.

8. Conclusions

Farsite is a scalable, decentralized, network file system wherein a loosely coupled collection of insecure and unreliable machines collaboratively establish a virtual file server that is secure and reliable. Farsite provides the shared namespace, location-transparent access, and reliable data storage of a central file server and also the low cost, decentralized security, and privacy of desktop workstations. It requires no central-administrative effort apart from signing user and machine certificates.

Farsite's core architecture is a collection of interacting, Byzantine-fault-tolerant replica groups, arranged in a tree that overlays the file-system namespace hierarchy. Because the vast majority of file-system data is opaque file content, Farsite maintains only indirection pointers and cryptographic checksums of this data as part of the Byzantine-replicated state. Actual content is encrypted and stored using raw (non-Byzantine) replication; however, the architecture could alternatively employ erasure-coded replication to improve storage efficiency.

Farsite is designed to support the file-I/O workload of desktop computers in a large company or university. It provides availability and reliability through replication; privacy and authentication through cryptography; integrity through Byzantine-fault-tolerance techniques; consistency through leases of variable granularity and duration; scalability through namespace delegation; and reasonable performance through client caching, hint-based pathname translation, and lazy update commit.

In large part, Farsite's design is a careful synthesis of techniques that are well known within the systems and security communities, including replication, Byzantine-fault-tolerance, cryptography, certificates, leases, client caching, and secret sharing. However, we have also developed several new techniques to address issues that have arisen in Farsite's design: Convergent encryption permits identifying and coalescing duplicate files encrypted with different users' keys. Exclusive encryption enforces filename syntax while maintaining filename privacy. A scalable, distributed, fault-tolerant database supports distributed duplicate-file detection. We use a novel combination of secret sharing, message authentication codes, and logging to enable secure crash recovery. Timed Byzantine operations and Byzantine outcalls supplement the conventional model of Byzantine fault-tolerance to permit directory groups to perform autonomous maintenance functions.

Analysis suggests that our design should be able to scale to our target of $\sim10^5$ machines. Experiments demonstrate that our untuned prototype provides tolerable performance relative to a local NTFS file system, and it performs significantly better than remote file access via CIFS.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, R. Wang. Serverless Network File Systems. *15th SOSP*, Dec 1995.

[2] R. J. Anderson, "The Eternity Service", *PRAGO-CRYPT '96*, CTU Publishing, Sep/Oct 1996.

[3] R. E. Blahut, *Theory and Practice of Error Control Codes*, Addison Wesley, 1983.

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, J. K. Ousterhout. "Measurements of a Distributed File System." 13th SOSP, Oct 1991.

[5] M. Blaze, "A Cryptographic File System for Unix", *1st Computer and Comm. Security*, ACM, Nov 1993.

[6] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs", *SIGMETRICS 2000*, ACM, Jun 2000.

[7] W. J. Bolosky, S. Corbin, D. Goebel, J. R. Douceur. Single Instance Storage in Windows 2000. *4th Usenix Windows System Symposium*, Aug 2000.

[8] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", *3rd OSDI*, USENIX, Feb 1999.

[9] I. Clarke, O. Sandberg, B. Wiley, and T. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System", *ICSI Workshop on Design Issues in Anonymity and Unobervability*, Jul 2000.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, "Wide-Area Cooperative Storage with CFS", *SOSP*, Oct 2001.

[11] J. R. Douceur, "The Sybil Attack", *1st IPTPS*, Mar 2002.

[12] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, M. Theimer, "Reclaiming Space from duplicate Files in a Serverless Distributed File System", *ICDCS*, Jul 2002.

[13] J. R. Douceur and W. J. Bolosky, "A Large-Scale Study of File-System Contents", *SIGMETRICS*, May 1999.

[14] J. R. Douceur and R. P. Wattenhofer, "Optimizing File Availability in a Secure Serverless Distributed File System", *20th SRDS*, IEEE, Oct 2001.

[15] J. R. Douceur, A. Adya; J. Benaloh; W. J. Bolosky; G. Yuval, "A Secure Directory Service based on Exclusive Encryption", (to appear) *18th ACSAC*, Dec 2002.

[16] J. Gemmell, E. M. Schooler, J. Gray, "Fcast Multicast File Distribution: 'Tune in, Download, and Drop Out'", *Internet, Multimedia Systems and Applications*, 1999.

[17] Gnutella, http://gnutelladev.wego.com.

[18] A. Goldberg and P. Yianilos, "Towards an Archival Intermemory", International Forum on Research and Technology Advances in Digital Libraries, Apr 1998.

[19] J. Gray. "Why do Computers Stop and What Can Be Done About It?", *5th SRDS*, Jan. 1986.

[20] J. Gray and A. Reuter, *Transaction Processiong: Concepts and Techniques*. Morgan Kaufmann, 1993.

[21] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, M. West, "Scale and Performance in a Distributed File System", *TOCS* 6(1), Feb 1988.

[22] J. Kistler, M. Satyanarayanan. Disconnected operation in the Coda File System. *TOCS* 10(1), Feb 1992.

[23] J. Kubiatowicz, et al., "OceanStore: An Architecture for Global-Scale Persistent Storage", *9th ASPLOS*, ACM, Nov 2000.

[24] E. Lee, C. Thekkath. Petal: Distributed virtual disks. *7th ASPLOS*, Oct 1996.

[25] M. Luby, "Benchmark Comparisons of Erasure Codes", http://www.icsi.berkeley.edu/~luby/erasure.html

[26] D. Mazières, M. Kaminsky, M. F. Kaashoek, E. Witchel, "Separating Key Management from File System Security", *SOSP*, Dec 1999.

[27] D. Mazières and D. Shasha, "Don't Trust Your File Server", 8th HotOS, May 2001.

[28] M. McKusick, W. Joy, S. Leffler, R. Fabry. A Fast File System for UNIX. *TOCS*, 2(3):181-197, Aug 1984.

[29] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1997.

[30] R. Merkle, "Protocols for Public Key Cryptosystems", *IEEE Symposium on Security and Privacy*, 1980.

[31] Microsoft, "File Name Conventions", *MSDN*, Apr 2002.

[32] Microsoft, "About Windows Update", http://v4.windowsupdate.microsoft.com/en/about.asp

[33] Napster, http://www.napster.com.

[34] E. Nemeth, G. Snyder, S. Seebass, T. R. Hein, *UNIX System Administration Handbook*, Prentice Hall, 2000.

[35] J. K. Ousterhout, A, R. Cherenson, F. Douglis, M. N. Nelson, B. B. Welch, "The Sprite Network Operating System", *IEEE Computer Group Magazine* 21 (2), 1988.

[36] M. Pease, R. Shostak, L. Lamport "Reaching Agreement in the Presence of Faults", *JACM* 27(2), Apr 1980.

[37] G. J. Popek, R. G. Guy, T. W. Page, J. S. Heidemann, "Replication in Ficus Distributed File Systems", IEEE Workshop on Management of Replicated Data, 1990.

[38] A. Rowstron and P. Druschel. "Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility", *SOSP*, Oct 2001.

[39] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon. Design and Implementation of the Sun Network File System. *Summer USENIX Proceedings*, 1985.

[40] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, J. Ofir, "Deciding When to Forget in the Elephant File System", *17th SOSP*, Dec 1999.

[41] S. T. Shafer, "The Enemy Within", *Red Herring*, Jan 2002.

[42] D. A. Solomon and M. E. Russinovich, *Inside Microsoft Windows 2000 Third Edition*, Microsoft Press, 2000.

[43] A. Sweeny, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, G. Peck, "Scalability in the XFS File System", *USENIX*, 1996.

[44] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System, *15th SOSP*, 1995.

[45] C. Thekkath, T. Mann, E. Lee. Frangipani: A Scalable Distributed File System. *16th SOSP*, Dec 1997.

[46] M. Tompa and H. Woll, "How to Share a Secret with Cheaters", *Journal of Cryptology* 1(2), 1998.

[47] S. Travaglia, P. Abrams, *Bastard Operator from Hell*, Plan Nine Publishing, Apr 2001.

[48] W. Vogels. File system usage in Windows NT 4.0. *17th SOSP*, Dec 1999.

[49] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kilite, P. Khosla, "Survivalbe Information Storage Systems", *IEEE Computer* 33(8), Aug 2000.

[50] B. Yee and J. D. Tygar, "Secure Coprocessors in Electronic Commerce Applications", *USENIX 95*, 1995.

Taming aggressive replication in the Pangaea wide-area file system

Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam
Storage Systems Department, HP Labs, Palo Alto, CA, USA
{ysaito,christos,karlsson,mmallik}@hpl.hp.com

Abstract

Pangaea is a wide-area file system that supports data sharing among a community of widely distributed users. It is built on a symmetrically decentralized infrastructure that consists of commodity computers provided by the end users. Computers act autonomously to serve data to their local users. When possible, they exchange data with nearby peers to improve the system's overall performance, availability, and network economy. This approach is realized by aggressively creating a replica of a file whenever and wherever it is accessed.

This paper presents the design, implementation, and evaluation of the Pangaea file system. Pangaea offers efficient, randomized algorithms to manage highly dynamic and potentially large groups of file replicas. It applies optimistic consistency semantics to replica contents, but it also offers stronger guarantees when required by the users. The evaluation demonstrates that Pangaea outperforms existing distributed file systems in large heterogeneous environments, typical of the Internet and of large corporate intranets.

1 Introduction

Pangaea is a wide-area file system that supports the daily storage needs of a distributed community of users. It is a platform for ad-hoc data sharing—it enables multinational corporations, distributed groups of collaborating users, and content management systems to exchange data efficiently using a file system.

Pangaea builds a unified file system across a federation of up to thousands of widely distributed computers connected by dedicated or virtual private networks. We currently assume that all servers are trusted; relaxing the trust relationship is future work. The system faces continuous reconfiguration, with users moving, companies restructuring, and computers being added or removed. Thus, Pangaea must meet three key goals:

Speed: Hide the wide-area networking latency; file access speed should resemble that of a local file system.

Availability and autonomy: Avoid depending on the availability of any specific node. Pangaea must adapt automatically to server additions, removals, failures and network partitioning.

Network economy: Minimize the use of wide-area networks. Nodes are not distributed uniformly; some nodes are in the same LAN, whereas some others are half way across the globe. Pangaea should transfer data between nodes in physical proximity, when possible, to reduce latency and save network bandwidth.

We argue that a system should follow a *symbiotic* design to achieve these goals in dynamic, wide-area environments. In such a system, each server functions autonomously and allows reads and writes to its files even when disconnected. As more computers become available, or as the system configuration changes, servers dynamically adapt and collaborate with each other, in a way that enhances the overall performance and availability of the system.

Pangaea realizes symbiosis by *pervasive replication*. It aggressively creates a replica of a file or directory whenever and wherever it is accessed. There is no single "master" replica of a file. Any replica may be read or written at any time, and replicas exchange updates among themselves in a peer-to-peer fashion. Pervasive replication achieves high performance by serving data from a server close to the point of access, high availability by letting each server contain its working set, and network economy by transferring data among close-by replicas. The following sections introduce two key strategies used to implement pervasive replication.

1.1 Graph-based replica management

Pangaea's replica management must satisfy three goals. First, it must support a large number of replicas, to maximize availability. Second, it needs to manage the replicas of each file independently, since it is difficult to predict file-access patterns accurately in a wide area. Third, it needs to support dynamic addition and removal of replicas even when some nodes are not available. Pangaea addresses these challenges by maintaining a sparse, yet strongly connected and randomized graph of replicas for each file. The graph is used both to propagate updates and to discover other replicas during replica addition and removal. This design offers three important benefits:

Available and inexpensive membership management: A replica can be added by connecting to a few live replicas that it discovers, no matter how many other replicas are unavailable. Since the graph is sparse, adding or removing a replica involves only a constant cost, regardless of the total number of replicas.

Available update distribution: Pangaea can distribute updates to all live replicas of a file as far as its graph is connected. The redundant and flexible nature of graphs makes them extremely unlikely to be disconnected even after multiple node or link failures.

Network economy: The random-graph design facilitates the efficient use of wide-area network bandwidth, for a system with an aggressive replication policy. Pangaea achieves this by clustering replicas in physical proximity tightly in the graph, and by creating a spanning tree along faster edges dynamically during update propagation.

1.2 Optimistic replica coordination

A distributed service faces two inherently conflicting challenges: high availability and strong data consistency [8, 37]. Pangaea aims at maximizing availability: at any time, users must be able to read and write any replica and the system must be able to create or remove replicas without blocking.

To address this challenge, Pangaea uses two techniques for replica management. First, it pushes updates to replicas rather than invalidating them, since the former achieves higher availability in a wide area by keeping up-to-date data in more locations. This approach may result in managing unnecessary replicas, wasting both storage space and networking bandwidth. To ameliorate this problem, Pangaea lets each node remove inactive replicas, as discussed in Section 4.4.

Second, Pangaea manages replica contents optimistically. It lets any node issue updates at any time, propagates them among replicas in the background, and detects and resolves conflicts after they happen. Thus, Pangaea supports only "eventual" consistency, guaranteeing that a user sees a change made by another user in some unspecified future time. Recent studies, however, reveal that file systems face very little concurrent write sharing, and that users demand consistency only within a window of minutes [31, 35]. Pangaea's actual window of inconsistency is around 5 seconds in a wide area, as we show in Section 7.6. In addition, Pangaea provides an option that synchronously pushes updates to all replicas and gives users confirmation of their update delivery (Section 5.3). We thus believe that Pangaea's consistency semantics are sufficient for the ad-hoc data sharing that Pangaea targets.

Pangaea does not support applications that require strong consistency such as open-close consistency, that use locks, or that synchronize using directory operations (i.e., "lock files").

2 Related work

Traditional local-area distributed file systems do not meet our goals of speed, availability, and network economy. Systems such as xFS [2] and Frangipani [33] rely on tight node coordination for replica management and cannot overcome the non-uniform networking latencies and frequent network partitioning that are typical in wide-area networks.

Pervasive replication resembles the persistent caching used in client-server file systems such as AFS [13], Coda [20], and LBFS [21]. Pangaea, however, can harness nodes to improve the system's robustness and efficiency. First, it provides better availability. When a server crashes, there are always other nodes providing access to the files it hosted. Updates can be propagated to all live replicas even when some of the servers are unavailable. The decentralized nature of Pangaea also allows any node to be removed (even permanently) transparently to users. Second, Pangaea improves efficiency by propagating updates between nearby nodes, rather than between a client and a fixed server and, creating new replicas from a nearby existing replica. In this sense, Pangaea generalizes the idea of Fluid replication [16] that utilizes surrogate Coda servers placed in strategic (but fixed) locations to improve the performance and availability of the system.

Pangaea's replication follows an optimistic approach similar to that of mobile data-sharing services, such as Lotus Notes [15], TSAE [10], Bayou [32], and Roam [25]. These systems lack replica location management and rely on polling, usually by humans, to discover and exchange updates between replicas. Pangaea keeps track of replicas automatically and distributes updates proactively and transparently to all the users. Most of these systems replicate at the granularity of the whole database (except Roam, which supports subset replicas). In contrast, Pangaea's files and directories are replicated independently, and some of its operations (e.g., "rename") affect multiple files, each replicated on a different set of nodes. Such operations demand a new protocol for ensuring consistent outcome after conflicts, as we discuss in Section 5.2. Pangaea offers a simple conflict resolution policy similar to that of Roam, Locus [36], or Coda [18]. We chose this design over more sophisticated approaches (as in Bayou), because Pangaea can make no assumptions about the semantics of file-system operations.

FARSITE [1] and Pangaea both build a unified file system across a federation of nodes, but they have different objectives. FARSITE's goal is to build a reliable service on top of untrusted nodes using Byzantine consensus protocols, and it is designed primarily for local-area networks. Pangaea assumes trusted servers, but it dynamically replicates files at the edge to minimize the use of wide-area networks.

Recent peer-to-peer data sharing systems, built on top of load-balanced, fault-tolerant distributed hash tables, share many properties with Pangaea. Systems such as CFS [6] and PAST [27] employ heuristics to exploit physical proximity when locating data, but they do not support concurrent in-place updates of hierarchically structured data. Pangaea, unlike these systems, provides extra machinery for conflict detection and resolution, as we discuss in Section 5.2. Oceanstore [17] builds a file system with strong consistency by routing updates through a small "core" of replicas. Pangaea, instead, allows in-place updating of any replica without centralized coordination to maximize availability. Ivy [22] is a peer-to-peer file system that lets data be updated, any time, anywhere, by exchanging operation logs between replicas. Because the replicas poll remote logs frequently, it supports stronger consistency than Pangaea. Its log-based update propagation also allows for more versatile conflict resolution than in Pangaea. However, because Ivy forces each user to read the logs of all other writers, it can only support a small file system with a small number of writers.

A number of companies are active in the field of wide-area collaborative data sharing, including FileFish, Scale8, WebFS, and Xythos. They offer a uniform, seamless interface for sharing files in a wide-area network, independent of the physical locations of users and data. Some of them provide features such as intelligent location of the cached copy closest to the user. However, they all use a centralized database to keep track of the location of files and replicas. Thus, their design does not meet Pangaea's goals of availability and autonomy.

3 Pangaea: a structural overview

This section overviews the structure of a server and the major data structures it maintains. Pangaea's design follows a symmetrically distributed approach. A Pangaea *server* handles file-access requests from users. We assume that a user uses a single server during a log-in session (lasting, say, a few hours), so that on-demand replication improves file-access latency; the user may move between servers over time. Each server maintains local hard disks, used to store replicas of files and directories. Servers interact with each other in a peer-to-peer fashion to provide a unified file-system.

3.1 Definitions

We use the terms *node* and *server* interchangeably. Nodes are automatically grouped into *regions*, such that nodes within a region have low round-trip times (RTT) between them (<5ms in our implementation). Pangaea uses region information to optimize replica placement and coordination.

Pangaea replicates data at the granularity of files and treats directories as files with special contents. Thus, we use the term *file* to refer to a regular file or a directory. An *edge* represents a known connection between two replicas of a file; updates to the file flow along edges. The replicas of a file and the edges between them comprise a strongly connected graph. The set of replicas of a file is called the file's *replica set*.

3.2 Structure of a server

The Pangaea server is currently implemented as a user-space NFSv3 loopback server (Figure 1). The server consists of four main modules:

NFS protocol handler receives requests from applications, updates local replicas, and generates requests for the replication engine. It is built using the SFS toolkit [19] that provides a basic infrastructure for NFS request parsing and event dispatching.

Replication engine accepts requests from the NFS protocol handler and the replication engine running on other nodes. It creates, modifies, or removes replicas, and forwards requests to other nodes if necessary. It is the largest part of the Pangaea server.

Log module implements transaction-like semantics for local disk updates via redo logging. The server logs all the replica-update operations using this service, allowing them to survive crashes.

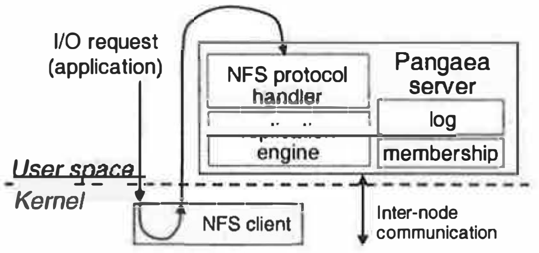


Figure 1: *The structure of the Pangaea server.*

Membership module maintains the status of other nodes, including their liveness, available disk space, the locations of root-directory replicas, the list of regions in the system, the set of nodes in each region, and a round-trip time (RTT) estimate between every pair of regions.

This module runs an extension of van Renesse's gossip-based protocol [34]. Each node periodically sends its knowledge of nodes' status to a random node chosen from its live-node list; the recipient merges this list with its own. A few fixed nodes are designated as "landmarks" and they bootstrap newly joining nodes. The protocol has been shown to disseminate membership information quickly with low probability of false failure detection.

The region and RTT information is gossiped as part of the membership information. A newly booted node obtains the region information from a landmark. It then polls a node in each existing region to determine where it belongs or to create a new singleton region. In each region, the node with the smallest IP address elects itself as a leader and periodically pings nodes in other regions to measure the RTT.

This membership-tracking scheme, especially the RTT management, is the key scalability bottleneck in our system—its network bandwidth consumption in a 10,000-node configuration is estimated to be 10K bytes/second/node. We plan to use external RTT-estimation services, such as IDMaps [9], once they become widely available.

3.3 Structure of a file system

Pangaea decentralizes both the replica-set and consistency management by maintaining a distributed graph of replicas for each file. Figure 2 shows an example of a system with two files. Pangaea distinguishes two types of replicas: *gold* and *bronze*. They can both be read and written by users at any time, and they both run an identical update-propagation protocol. Gold replicas, however, play an additional role in maintaining the hierarchical name space.

First, gold replicas act as starting points from which bronze replicas are found during path-name traversal. To this end, the directory entry of a file lists the file's gold replicas. Second, gold replicas perform several tasks that are hard to do in a completely distributed way. In particular, they are used as pivots to keep the graph connected after a permanent node failure, and to maintain a minimum replication factor for a file. They form a clique in the file's graph so that they can monitor each other for these tasks. These issues are discussed in more detail in Section 4. Currently, Pangaea designates replicas created during initial file creation as gold and fixes their locations unless some of them fail permanently.

Each replica stores a *backpointer* that indicates its location in the file-system name space. A backpointer includes the parent directory's ID and the file's name within the directory.[1] It is used for two purposes: to resolve conflicting directory operations (Section 5.2), and to keep the directory entry up-to-date when the gold replica set of the file changes (Section 6.2).

Figure 3 shows the key attributes of a replica. The timestamp (*ts*) and the version vector (*vv*) [23] record the last time the file was modified. Their use is described in more detail in Section 5. *GoldPeers* are uni-directional links to the gold replicas of the file. *Peers* point to the neighboring (gold or bronze) replicas in the file's graph.

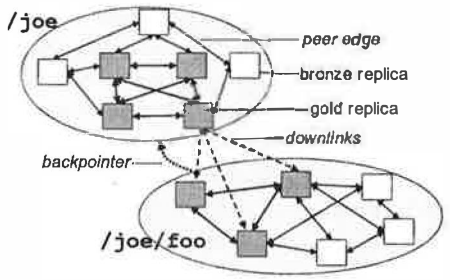


Figure 2: *An example of a directory `/joe` and file `/joe/foo`. Each replica of `joe` stores three pointers to the gold replicas of `foo`. Each replica of `foo` keeps a backpointer to the parent directory. Bronze replicas are connected randomly to form strongly connected graphs. Bronze replicas also have uni-directional links to the gold replicas of the file, which are not shown here.*

4 Replica set management

In Pangaea, a replica is created when a user first accesses the file, and it is removed when a node runs out of disk space or finds a replica to be inactive. Because these operations are frequent, they must be carried out efficiently and

[1] A replica stores multiple backpointers when the file is hard-linked. A backpointer need not remember the locations of the parent-directory replicas, since a parent directory is always found on the same node due to the namespace-containment property (Section 4.3).

```
struct Replica
    fid: FileID                  // 96 bit globally unique file ID
    ts: TimeStamp                // Pair of ⟨physical clock, IP addr⟩.
    vv: VersionVector            // Maps IP addr ↦ TimeStamp
    goldPeers: Set⟨NodeID⟩ // Set of IP addresses
    peers: Set⟨NodeID⟩
    backptrs: Set⟨FileID, String⟩ // Pair of ⟨dirID, fname⟩
    ...
end
struct DirEntry
    fname: String
    fid: FileID
    downlinks: Set⟨NodeID⟩
    ts: Timestamp
end
```

Figure 3: *Key attributes of a replica.*

without blocking, even when some nodes that store replicas are unavailable. This section describes algorithms based on random walks that achieve these goals.

4.1 File creation

We describe the interactions between the modules of the system and the use of various data structures using a simple scenario—a user on server S creates file F in directory D.

For the moment, assume that S already stores a replica of D (if not, S creates one, using the protocol described in Section 4.2.) First, S determines the location of g initial replicas of F, which will become the gold replicas of the file (a typical value for g is 3). One replica will reside on S. The other $g-1$ replicas are chosen at random from different regions in the system to improve the expected availability of the file. Second, S creates the local replica for F and adds an entry for F in the local replica of D. S then replies to the client, and the client can start accessing the file.

In the background, S disseminates two types of updates. It first "floods" the new directory contents to other directory replicas. It also floods the contents of F (which is empty, save for attributes such as permissions and owner) to its gold-replica nodes. In practice, as we describe in Section 5, we deploy several techniques to reduce the overhead of flooding dramatically. As a side effect of the propagation, the replicas of D will point to F's gold replicas so that the latter can be discovered during future path-name lookups.

4.2 Replica addition

The protocol for creating additional replicas for a file is run when a user tries to access a file not present in her local node. Say that a user on node S wants to read file F. A read or write request is always preceded by a directory lookup (during the `open` request) on S. Thus, to create a replica, S must replicate the file's parent directory. This recursive step may continue all the way up to the root directory. The locations of root replicas are maintained by the membership service (Section 3.2).

Pangaea performs a *short-cut* replica creation to transfer data from a nearby existing replica. To create a replica of F, S first discovers the file's gold replicas in the directory entry during the path-name lookup. S then requests the file contents from the gold replica closest to S (say P). P then finds a replica closest to S among its own graph neighbors (say X, which may be P itself) and forwards the request to X, which in turn sends the contents to S. At this point, S replies to the user and lets her start accessing the replica. This request forwarding is performed because the directory only knows F's gold replicas, and there may be a bronze replica closer to P than the gold ones.

The new copy must be integrated into the file's replica graph to be able to propagate updates to and receive updates from other replicas. Thus, in the background, S chooses m existing replicas of F, adds edges to them, and requests them to add edges to the new replica in S. The selection of m peers must satisfy three goals:

- Include gold replicas so that they have more choices during future short-cut replica creation.
- Include nearby replicas so that updates can flow through fast network links.
- Be sufficiently randomized so that, with high probability, the crash of nodes does not catastrophically disconnect the file's graph.

Pangaea satisfies all these goals simultaneously, as a replica can have multiple edges. S chooses three types of peers for the new replica. First, S adds an edge to a random gold replica, preferably one from a different region than S, to give that gold replica more variety of regions in its neighbor set. Second, it asks a random gold replica, say P, to pick the replica (among P's immediate graph neighbors) closest to S. Third, S asks P to choose $m-2$ random replicas using random walks that start from P and perform a series of RPC calls along graph edges. This protocol ensures that the resulting graph is m edge- and node- connected, provided that it was m-connected before.

Parameter m trades off availability and performance. A small value increases the probability of graph disconnection (i.e., the probability that a replica cannot exchange updates with other replicas) after node failures. A large value for m increases the overhead of graph maintenance and update propagation by causing duplicate update delivery. We found that $m = 4$ offers a good balance in our prototype.

4.3 Name-space containment

The procedures for file creation and replica addition both require a file's parent directory to be present on the same node. Pangaea, in fact, demands that for every file, all intermediate directories, up to the root, are always replicated on the same node. This *name-space-containment* requirement yields two benefits. First, it naturally offers the availability and autonomy benefits of island-based replication [14]. That is, it enables lookup and access to every replica even when the server is disconnected and allows each node to take a backup of the file system locally. We quantify these benefits in Section 7.8. Second, it simplifies the conflict resolution of directory operations, as we discuss in Section 5.2.

On the other hand, this requirement increases the system-wide storage overhead by 1.5% to 25%, compared to an idealized scheme in which directories are stored on only one node [28].[2] We consider the overhead to be reasonable, as users already pay many times more storage cost by replicating files in the first place.

4.4 Bronze replica removal

This section describes the protocol for removing bronze replicas. Gold replicas are removed only as a side effect of a permanent node loss. We discuss the handling of permanent failures in Section 6.

A replica is removed for two possible reasons: because a node has run out of disk space, or the cost of keeping the replica outweighs its benefits. To reclaim disk space, Pangaea uses a randomized GD-Size algorithm [24]. We examine 50 random replicas kept in the node and calculate their merit values using the GD-Size function that considers both the replica's size and the last-access time [5]. The replica with the minimum merit is evicted, and five replicas with the next-worst merit values are added to the candidates examined during the next round. The algorithm is repeated until it frees enough space on the disk.

Optionally, a server can also reclaim replicas not worth keeping. We currently use a competitive updates algorithm for this purpose [12]. Here, the server keeps a per-replica counter that is incremented every time a replica receives a remote update and is reset to zero when the replica is read. When the counter's value exceeds a threshold (4 in our prototype), the server evicts the replica.

[2] Due to the lack of wide-area file system traces, we analyzed the storage overhead using a fresh file system with RedHat 7.3 installed. The overhead mainly depends on the spatial locality of accesses, i.e., the degree to which files in the same directory are accessed together. We expect the overhead in practice to be much closer to 1.5% than 25%, because spatial locality in typical file-system traces is usually high.

To remove a replica, the server sends notices to the replica's graph neighbors. Each neighbor, in turn, initiates a random walk starting from a random gold replica[3] and uses the protocol described in Section 4.2 to establish a replacement edge with another live replica. Starting the walk from a live gold replica ensures that the graph remains strongly connected. A similar protocol runs when a node detects another node's permanent death, as we describe in Section 6.

4.5 Summary

The graph-based pervasive replication algorithms described in this section offer some fundamental benefits over traditional approaches that have a fixed set of servers manage replica locations.

Simple and efficient recovery from failures: Graphs are, by definition, flexible—spanning edges to *any* replica makes the graph incrementally more robust and efficient. Moreover, using just one type of edges both to locate replicas and to propagate updates simplifies the recovery from permanent failures and avoids any system disruption during graph reconfiguration.

Decoupling of directories and files: Directory entries point only to gold replicas, and the set of gold replicas is typically stable. Thus, a file and its parent directory act mostly independently once the file is created. Adding or removing a bronze replica for the file does not require a change to the directory replicas. Adding or removing a gold or bronze replica for the directory does not require a change to the file replicas. These are key properties for the system's efficiency.

5 Propagating updates

This section describes Pangaea's solutions to three challenges posed by optimistic replication: efficient and reliable update propagation, handling concurrent updates, and the lack of strong consistency guarantees.

5.1 Efficient update flooding

The basic method for propagating updates in Pangaea is *flooding* along graph edges, as shown in Figure 4. Whenever a replica is modified on a server, the server pushes the entire file contents to all the graph neighbors, which in turn forward the contents to their neighbors, and so on, until all the replicas receive the new contents. This simple flooding

[3]The gold-replica set is kept as a part of the replica's attributes; see Figure 3.

```
r: Replica being updated.
when Update is newly issued
  Log ⟨r.fid, r.vv⟩.
  Send ⟨r.fid, r.vv, r.data⟩ to nodes in r.peers
  Unlog ⟨r.fid, r.vv⟩ after all the neighbors reply.
when Update ⟨fid, vv, data⟩ is received from node n.
  if this update has already been applied then Reply to n
  else Log and apply the update.
       Reply to n
       Forward the update to r.peers - {n}.
       Unlog after all the neighbors reply.
```

Figure 4: *A simple flooding algorithm to distribute updates. This code assumes that updates are issued one at a time; the handling of concurrent updates is discussed in Section 5.2.*

algorithm guarantees reliable update delivery as long as the replica graph is strongly connected. The following three sections introduce techniques for improving the efficiency of the basic flooding algorithm.

5.1.1 Optimization 1: delta propagation

A major drawback of flooding is that it propagates the entire file contents even when only one byte has been modified. Delta propagation improves the propagation efficiency while maintaining the logical simplicity of flooding. Here, whenever a portion of a file is changed (e.g., adding an entry to a directory), Pangaea propagates only a small, semantic description of the change, called a *delta*. Deltas, in general, must be applied in the same order to every replica to produce the same result. We ensure this by having each delta carry two timestamps: the *old timestamp* that represents the state of the replica just before the change, and the *new timestamp* that shows the state of the replica after the change [15]. A replica applies a delta only when its current timestamp matches the delta's old timestamp. Otherwise, it resorts to full contents transfer, with potential conflict resolution as described in Section 5.2. In practice, updates are handled almost exclusively by deltas, and full-state transfer happens only when there are concurrent writes, or when a node recovers from a crash.

Pangaea further reduces the size of updates by delta merging, akin to the feature implemented in Coda [20]. For example, when a file is deleted right after it is modified (which happens often for temporary files), the server quashes the modification if it has not yet been sent to other replicas. Delta merging is transparent to users because it adds no delay to propagation.

5.1.2 Optimization 2: harbingers

Flooding guarantees reliable delivery by propagating updates (deltas or full contents) over multiple links at each step of the algorithm. Thus, it consumes m times the optimal network bandwidth, where m is the number of edges per replica. Harbingers eliminate redundant update deliveries.

Pangaea uses a two-phase protocol to propagate updates that exceed a certain size (1KB). In phase one, a small message that only contains the timestamps of the update, called a *harbinger*, is flooded along graph edges. The update bodies are sent, in phase two, only when requested by other nodes. When a node receives a new harbinger, it asks the sender of the harbinger (the immediate upstream replica in the flooding chain) to push the update body. Simultaneously, it forwards the harbinger to other neighbors in the graph. When a node receives a duplicate harbinger without having received the update body, it asks its sender to retry later. This is required because the sender of the earliest harbinger may crash before sending the update body. If a node receives a harbinger after having received the update body, it tells the sender to stop sending the update. We chose the harbinger threshold of 1KB, because we found that delta sizes follow a bimodal distribution—one peak around 200 bytes representing directory operations, and a flatter plateau around 20KB representing bulk writes.[4]

This harbinger algorithm not only saves network usage, but also shrinks the effective window of replica inconsistency. When a user tries to read a file for which only a harbinger has been received, she waits until the actual update arrives. Since harbinger-propagation delay is independent of the actual update size, the chance of a user seeing stale file contents is greatly reduced.

5.1.3 Optimization 3: exploiting physical topology

Harbingers have another positive side effect. They favor the use of fast links, because a node requests the body of an update from the sender of the first harbinger it receives. However, unpredictable node or link load may reduce this benefit. A simple extension to the harbinger algorithm improves the data propagation efficiency, without requiring any coordination between nodes. Before pushing (or forwarding) a harbinger over a graph edge, a server adds a delay proportional to the estimated speed of the edge (10$*$RTT in our implementation). This way, Pangaea dynamically builds a spanning tree whose shape closely matches the physical network topology. Figure 5 shows an example. In Section 7.6, we show that this technique drastically reduces the use of wide-area networks when updating shared files.

[4]Pangaea batches NFS write requests and flushes data to disk and other replicas only after a "commit" request [4]. Thus, the size of an update can grow larger than the typical "write" request size of 8KB.

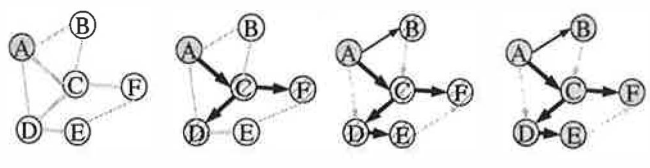


Figure 5: *An example of update propagation for a file with six replicas, A to F. Thick edges represent fast links. (1) An update is issued at A. (2) A sends a harbinger via the fat edge to C. C forwards the harbinger to D and F quickly. (3) D forwards the harbinger to E. After some time, A sends the harbinger to B, and a spanning tree is formed. Links not in the tree are used as backups when some of the tree links fail. (4) The update's body is pushed along the tree edges. In practice, steps 2-4 proceed in parallel.*

5.2 Conflict resolution

With optimistic replication, concurrent updates are inevitable, although rare [35, 31]. We use a combination of version vectors and the last-writer-wins rule to resolve conflicts.

First, recall that when delta timestamps mismatch, servers revert to full-state transfer. We then use version vectors [23] to separate true conflicts from other causes (e.g., missing updates) that can be fixed simply by overwriting the replica. This simplifies conflict resolution.

For conflicts on the contents of a regular file, we currently offer users two options. The first is the "last-writer-wins" rule using update timestamps (attribute *ts* in Figure 3). In this case, the clocks of servers should be loosely synchronized, e.g., using NTP, to respect the users' intuitive sense of update ordering. The second option is to concatenate two versions in the file and let the user fix the conflict manually. Other options, such as application-specific resolvers [36, 18, 32], are certainly possible, but we have not implemented them yet.

Conflicts regarding file attributes or directory entries are more difficult to handle. They fall into two categories. The first is a conflict between two directory-update operations; for example, Alice does "`mv /foo /alice/foo`" and Bob does "`mv /foo /bob/foo`" concurrently. In the end, we want one of the updates to take effect, but not both. The second category is a conflict between "rmdir" and any other operation; for example, Alice does "`mv /foo /alice/foo`" and Bob does "`rmdir /alice`". These problems are difficult to handle, because files may be replicated on different sets of nodes, and a node might receive only one of the conflicting updates and fail to detect the conflict in the first place.

We only outline our solution here, as it is fully described in [28]. Our principle is always to let the child file ("`foo`" in our example), rather than its parent ("`alice`" or "`bob`"), dictate the outcome of the conflict resolution using the "last-writer-wins" rule. We thus let the file's backpointer (Section 3.3) *authoritatively* define the file's location in the file-system namespace. We implement directory operations, such as "mv" and "rm", as a change to the file's backpointer(s). When a replica receives a change to its backpointer, it also reflects the change to its parents by creating, deleting, or modifying the corresponding entries.[5] The parent directory will, in turn, flood the change to its replicas. In practice, we randomly delay the directory-entry patching and subsequent flooding, because there is a good chance that other replicas of the file will do the same. Figure 6 illustrates how Pangaea resolves the first conflict scenario. The same policy is used to resolve the mv-rmdir conflict: when a replica detects the absence of the directory entry corresponding to its backpointer, it re-creates the entry, which potentially involves re-creating the directory itself and the ancestor directories recursively, all the way to the root.

A directory in Pangaea is, in effect, merely a copy of the backpointers of its children. Thus, resolving conflicts on directory contents is done by applying the "last-writer-wins" rule to individual entries. If a file is to be removed from a directory, the directory still keeps the entry but marks it as "dead" (i.e., it acts as a "death certificate" [7]), so that we can detect when a stale change to the entry arrives in the future.

5.3 Controlling replica divergence

The protocols described so far do not provide hard guarantees for the degree of replica divergence—consistency is achieved only eventually.

To alleviate this problem, Pangaea introduces an option, called the "red button", to provide users confirmation of update delivery. The red button, when pressed for a particular file, sends harbingers for any pending updates to neighboring replicas. These harbingers (and corresponding updates) circulate among replicas as described in Section 5.1.2. A replica, however, does not acknowledge a harbinger until all the graph neighbors to which it forwarded the harbinger acknowledge it or time out (to avoid deadlocking, a replica replies immediately when it receives the same harbinger twice). The user who pressed the red button waits until the operation is fully acknowledged or some replicas time out, in which case the user is presented with the list of unavailable replicas.

[5] The replica can always find a replica of the parent directory in the same node, because of the name-space-containment property.

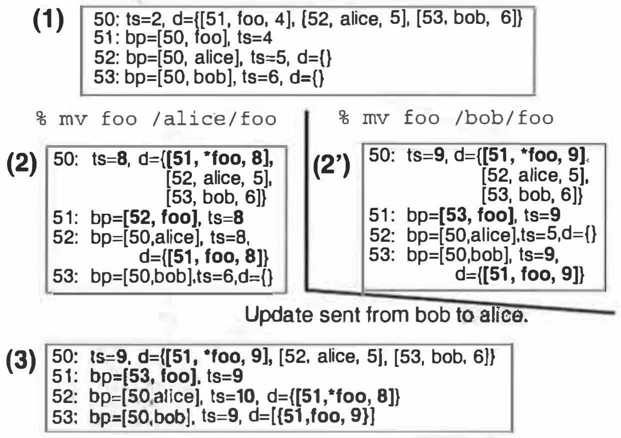

(1)
50: ts=2, d={[51, foo, 4], {52, alice, 5], [53, bob, 6]}
51: bp=[50, foo], ts=4
52: bp=[50, alice], ts=5, d={}
53: bp=[50, bob], ts=6, d={}
% mv foo /alice/foo
% mv foo /bob/foo
(2)
50: ts=8, d={[51, *foo, 8], [52, alice, 5], [53, bob, 6]}
51: bp=[52, foo], ts=8
52: bp=[50,alice], ts=8, d={[51, foo, 8]}
53: bp=[50,bob],ts=6,d={}
(2')
50: ts=9, d={[51, *foo, 9], [52, alice, 5], [53, bob, 6]}
51: bp=[53, foo], ts=9
52: bp=[50,alice],ts=5,d={}
53: bp=[50,bob], ts=9, d={[51, foo, 9]}
Update sent from bob to alice.
(3)
50: ts=9, d={[51, *foo, 9], [52, alice, 5], [53, bob, 6]}
51: bp=[53, foo], ts=9
52: bp=[50,alice], ts=10, d={[51,*foo, 8]}
53: bp=[50,bob], ts=9, d=[{51,foo, 9}]

Figure 6: *Example of conflict resolution involving four files, "/" (FileID=50), "/foo" (FileID=51), "/alice/" (FileID=52), and "/bob/" (FileID=53). "ts=2" shows the replica's timestamp. "bp=[50,foo]" shows that the backpointer of the replica indicates that the file has the name "foo" in the directory 50 ("/"). "d={[51,foo,4]}" means that the directory contains one entry, a file "foo" with ID of 51 and timestamp of 4. Bold texts indicate changes from the previous step. Entries marked "*foo" are death certificates. (1) Two sites initially store the same contents. (2) Alice does "mv /foo /alice/foo". (2') Bob concurrently does "mv /foo /bob/foo" on another node. Because Bob's update has a newer timestamp (ts=9) than Alice's (ts=8), we want Bob's to win over Alice's. (3) When Alice's node receives the update from Bob's, the replica of file 51 will notice that its backpointer has changed from [52, foo] to [53, foo]. This change triggers the replica to delete the entry from `/alice` and add the entry to `/bob`.*

This option gives the user confirmation that her updates have been delivered to remote nodes and allows her to take actions contingent upon stable delivery, such as emailing her colleagues about the new contents. The red button, however, still does not guarantee a single-copy serializability, as it cannot prevent two users from changing the same file simultaneously.

6 Failure recovery

Failure recovery in Pangaea is simplified due to three properties: 1) the randomized nature of replica graphs that tolerate operation disruptions; 2) the idempotency of update operations; including NFS requests; and 3) the use of a unified logging module that allows any operation to be re-started.

We distinguish two types of failures: temporary failures and permanent failures. They are currently distinguished simply by their duration—a crash becomes permanent when a node is suspected to have failed continuously for more than two weeks. Given that the vast majority of failures are temporary [11, 3], we set two different goals. For temporary failures, we try to reduce the recovery cost. For permanent failures, we try to clean all data structures associated with the failed node so that the system runs as if the node had never existed in the first place.

6.1 Recovering from temporary failures

Temporary failures are handled by retrying. A node persistently logs any outstanding remote-operation requests, such as contents update, random walk, or edge addition. A node retries logged updates upon reboot or after it detects another node's recovery. This recovery logic may sometimes create uni-directional edges or more edges than desired, but it maintains the most important invariant, that the graphs are *m*-connected and that all replicas are reachable in the hierarchical name space.

Pangaea reduces the logging overhead during contents-update flooding, by logging only the ID of the modified file and keeping deltas only in memory. To reduce the memory footprint further, when a node finds out that deltas to an unresponsive node are piling up, the sender discards the deltas and falls back on full-state transfer.

6.2 Recovering from permanent failures

Permanent failures are handled by a garbage collection (GC) module. The GC module periodically scans local disks and discovers replicas that have edges to permanently failed nodes. When the GC module finds an edge to a failed bronze replica, it replaces the edge by performing a random walk starting from a gold replica (Section 4.4).

Recovering from a permanent loss of a gold replica is more complex. When a gold replica, say *P*, detects a permanent loss of another gold replica, *P* creates a new gold replica on a live node chosen using the criteria described in Section 4.1. Because gold replicas form a clique (Section 3.3), *P* can always detect such a loss. This choice is flooded to *all* the replicas of the file, using the protocol described in Section 5, to let them update their uni-directional links to the gold replicas. Simultaneously, *P* updates the local replica of the parent directory(ies), found in its backpointer(s), to reflect *P*'s new gold-replica set. This change is flooded to other replicas of the directories. Rarely, when the system is in transient state, multiple gold replicas may initiate this protocol simultaneously. Such a situation is resolved using the last-writer-wins policy, as described in Section 5.2.

Recovering from a permanent node loss is an inherently expensive procedure, because data stored on the failed node must eventually be re-created somewhere else. The problem is exacerbated in Pangaea, because it does not have a central authority to manage the locations of replicas—

all surviving nodes must scan their own disks to discover replicas that require recovery. To lessen the impact, the GC module tries to discover as many replicas that needs recovery as possible with a single disk scan. We set the default GC interval to be every three nights, which reduces the scanning overhead dramatically while still offering the expected file availability in the order of six-nines, assuming three gold replicas per file and a mean server lifetime of 290 days [3].

7 System evaluation

This section evaluates the design and implementation of Pangaea. First, we investigate the baseline performance and overheads of Pangaea and show that it performs competitively with other distributed file systems, even in a LAN. Further, we measure the latency, network economy, and availability of Pangaea in a wide-area networking environment in the following ways:

- We study the latency of Pangaea using two workloads: a personal workload (Andrew benchmark) and a BBS-like workload involving extensive data sharing. For the personal workload, we show that the user sees only local access latency on a node connected to a slow network and that roaming users can benefit by fetching their personal data from nearby sources. Using the second workload, we show that as a file is shared by more users, Pangaea progressively lowers the access latency by transferring data between nearby clients.
- We demonstrate network economy by studying how updates are propagated for widely shared files. We show that Pangaea transfers data predominantly over fast links.
- To demonstrate the effect of pervasive replication on the availability of the system, we analyze traces from a file server and show that Pangaea disturbs users far less than traditional replication policies.

7.1 Prototype implementation

We have implemented Pangaea as a user-space NFS (version 3) server using the SFS toolkit [19]. Our prototype implements all the features described in the paper, except that support for recovery from permanent failures (Section 6) is still fragmentary. Pangaea currently consists of 30,000 lines of C++ code.

A Pangaea server maintains three types of files on the local file system: *data* files, the *metadata* file, and the *intention-log* file. A data file is created for each replica of a file or directory. The node-wide metadata file keeps the extended attributes of all replicas stored on the server, including graph edges and version vectors. Data files for directories and the metadata file are both implemented using the Berkeley DB library [30] that maintains a hash table in a file. The intention-log file is also implemented using the Berkeley DB to record update operations that must survive a node crash. All the Berkeley DB files are managed using its "environments" feature that supports transactions through low-level logging. This architecture allows metadata changes to multiple files to be flushed with a sequential write to the low-level log.

Type	#	CPU	Disk	Mem
A	2	730MHz	Quantum Atlas 9WLS	256MB
B	3	1.8GHz	Quantum Atlas TW367L	512MB
C	4	400MHz	Seagate Cheetah 39236LW	256MB

Table 1: *The type and number of PCs used in the experiments. All the CPUs are versions of Pentiums.*

7.2 Experimental settings

We compare Pangaea to Linux's in-kernel NFS version 3 server and Coda, all running on Linux-2.4.18, with ext3 as the native file system.

We let each Pangaea server serve only clients on the same node. Both Pangaea and NFS flush buffers synchronously to disk before replying to a client, as required by the NFS specifications [4]. Coda supports two main modes of operation: strongly connected mode (denoted *coda-s* hereafter) that provides open-close semantics, and weakly connected mode (denoted *coda-w* hereafter) that improves the response-time of write operations by asynchronously trickling updates to the server. We mainly evaluate coda-w, since its semantics are closer to Pangaea's.

Table 1 shows the machines we used for the evaluation. All the machines are physically connected by a 100Mb/s Ethernet. Disks on all the machines are large enough that replicas never had to be purged in either Pangaea or Coda. For NFS and Coda, we configured a single server on a type-A machine. Other machines are used as clients. For Pangaea, all machines are used as servers and applications access files from their local servers. For CPU-intensive workloads (i.e., Andrew), we used a type-A machine for all the experiments. The other experiments are completely network-bound, and thus they are insensitive to CPU speeds.

For our wide-area experiments, we built a simulated WAN to evaluate Pangaea reliably in a variety of networking conditions. We routed packets to a type-B FreeBSD node (not included in the table) running Dummynet [26] to add artificial delays and bandwidth restrictions. This router node was fast enough never to become a bottleneck in any of our experiments.

7.3 Baseline performance in a LAN

This section evaluates Pangaea's performance in a LAN using a sequential workload without data sharing. While such an environment is not Pangaea's main target, we conducted this study to test Pangaea's ability to serve people's daily storage needs and to understand the system's behavior in an idealized situation.

We created a variation of the Andrew benchmark[6] that simulates a single-person, engineering-oriented workload. It has the same mix of operations as the original Andrew benchmark [13], but the volume of the data is expanded twenty-fold to allow for accurate measurements on modern hardware. This benchmark, denoted *Andrew-Tcl* hereafter, consists of five stages: (1) *mkdir*: creating 200 directories, (2) *copy*: copying the Tcl-8.4 source files from one directory to another, (3) *stat*: doing "ls -l" on the source files, (4) *grep*: doing "du" and "grep" on the source files, and (5) *compile*: compiling the source code. We averaged results from four runs per system, with 95% confidence interval below 3% for all the numbers presented.

Table 2 shows the time to complete the benchmark. Throughout the evaluation, label pang-N stands for a Pangaea system with N (gold) replicas per file. Pangaea's performance is comparable to NFS. This is as expected, because both systems perform about the same amount of buffer flushing, which is the main source of overhead. Pangaea is substantially slower only in `mkdir`. This is because Pangaea must create a Berkeley DB file for each new directory, which is a relatively expensive operation. Pangaea's performance is mostly independent of a file's replication factor, thanks to optimistic replication, where most of the replication processing happens in the background.

Coda's weakly connected mode (coda-w) is very fast. This is due to implementation differences: whereas Pangaea and NFS flush buffers to disk after every update operation, Coda avoids that by intercepting low-level file-access (VFS) requests using a small in-kernel module.

Figure 7 shows the network bandwidth used during the benchmark. "Overhead" is defined to be harbingers and update messages that turn out to be duplicates. Pang-1 does not involve any network activity since it stores files only on the local server. Numbers for pang-3 and -4 show the effect of Pangaea's harbinger algorithm in conserving network-bandwidth usage. In this benchmark, because all

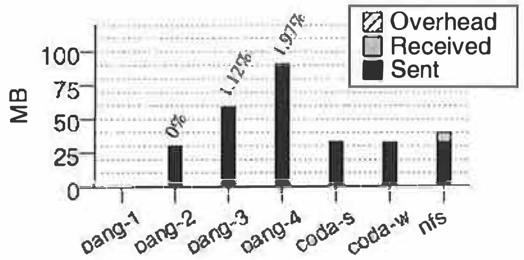


Figure 7: *Network bandwidth consumed during the Andrew benchmark. The "overhead" bars show bytes consumed by harbingers and duplicate updates. The numbers above the bars show the percentage of overhead.*

replicas are gold and they form a clique, Pangaea would have consumed 4 to 9 times the bandwidth of pang-2 were it not for harbingers. Instead, its network usage is near-optimal, with less than 2% of the bandwidth wasted.

Table 3 shows network bandwidth consumption for common file-system update operations. Operations such as creating a file or writing one byte show a high percentage of overhead, since they are sent directly without harbingers, but they have only a minor impact on the overall wasted bandwidth since their size is small. On the other hand, bulk writes, which make up the majority of the overall traffic, incur almost no overhead.

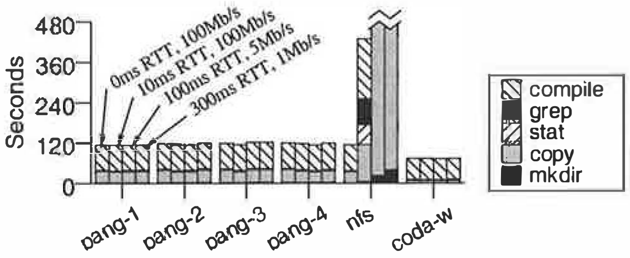


Figure 8: *Andrew-Tcl benchmark results on a node with a slow network link. The labels next to the bars indicate the link speeds. For Pangaea, these are the links between any two servers; for NFS and Coda, they are the links between clients and server. NFS took 1939 seconds in a 5Mb/s network, and it did not finish after two hours in a 1Mb/s network.*

7.4 Performance of personal workload in WANs

We ran the Andrew-Tcl benchmark to study the performance of the systems in WANs for a personal workload. Since this workload involves no data sharing, the elapsed time depends (if at all) only on the latency and capacity of the link between the client and the server. Figure 8 shows the time needed to complete the benchmark. Pangaea and

[6]This benchmark is available from `http://www.hpl.hp.com/personal/ysaito`.

	pang-1	pang-2	pang-3	pang-4	NFS	Coda-s	Coda-w	ext3
mkdir	2.04	2.04	2.18	2.28	0.316	2.25	0.047	0.021
copy	3.40	3.79	3.85	3.90	3.50	201.0	0.85	0.264
stat	0.91	0.90	0.90	0.91	0.87	0.86	0.86	0.162
grep	2.09	2.11	2.13	2.13	2.20	1.22	1.20	0.925
compile	74.4	75.3	75.8	75.9	77.2	90.2	62.1	61.5
Total	82.84	84.14	84.86	85.12	84.08	295.5	65.05	62.87

Table 2: *Andrew-Tcl benchmark results in a LAN environment. Numbers are in seconds. Label pang-N shows Pangaea's performance when it creates N replicas for each new file. Ext3 is Linux's native (local) file system.*

	pang-1	pang-2		pang-3		pang-4		NFS	coda-w	coda-s
	Bytes	Bytes	Overhead	Bytes	Overhead	Bytes	Overhead	Bytes	Bytes	Bytes
create	0	248	0%	1.29K	60%	2.61K	68%	503	1.46K	1.96K
write 1B	0	323	0%	854	61%	2.01K	68%	667	944	935
write 50KB	0	52.04K	0%	104.98K	1.49%	157.44K	1.52%	53.21K	55.56K	82.13K
write 25MB	0	26.22M	0%	52.44M	0.01%	78.67M	0.02%	26.76M	1.56M	38.75M

Table 3: *Network bandwidth consumption for common file-system operations. Shows the total number of bytes transmitted between all the nodes for each operation. "Overhead" shows the percentage of the bandwidth used by harbingers and duplicate updates.*

Coda totally hide the network latency, because the benchmark is designed so that it reads all the source data from the local disk, and the two systems can propagate updates to other nodes in the background. On the other hand, the performance of NFS degrades severely across slow links.

7.5 Roaming

Roaming, i.e., a single user moving between different nodes, is an important use of distributed file systems. We expect Pangaea to perform well in non-uniform networks in which nodes are connected with networks of different speeds. We simulated roaming using three nodes: S, which stores the files initially and is the server in the case of Coda, and two type-A nodes, C_1 and C_2. We first run the Andrew-Tcl benchmark to completion on node C_1, delete the `*.o` files, and then re-run only the compilation stage of the benchmark on node C_2. We vary two parameters: the link speed between C_1 and C_2, and the link speed between them and S. As seen from Figure 8, the performance depends, if at all, only on these two parameters.

Figure 9 shows the results. It shows that when the network is uniform, i.e., when the nodes are placed either all close by or all far apart, Pangaea and Coda perform comparably. However, in non-uniform networks, Pangaea achieves better performance than Coda by transferring data between nearby nodes. In contrast, Coda clients always fetch data from the server. (Pangaea actually performs slightly better in uniformly slow networks. We surmise that the reason is that Pangaea uses TCP for data transfer, whereas Coda uses its own UDP-based protocol.)

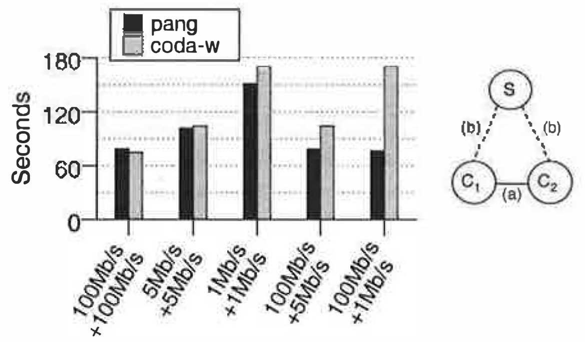


Figure 9: *The result of recompiling the Tcl source code. 100Mb/s + 1Mb/s, for example, means that the link between the two client nodes (link (a) in the right-side picture) is 100Mb/s, and the link between the benchmark client and the server (link (b)) is 1Mb/s. The speed of other links is irrelevant in this experiment.*

7.6 Data sharing in non-uniform environments

The workload characteristics of wide-area collaboration systems are not well known. We thus created a synthetic benchmark modeled after a bulletin-board system. In this benchmark, articles (files) are continuously posted or updated from nodes chosen uniformly at random; other randomly chosen nodes (i.e., users) fetch new articles not yet read. A file system's performance is measured by two metrics: the mean latency of reading a file never accessed before by the server, and the wide-area network bandwidth consumption for files that are updated. These two numbers depend, if at all, only on the file size, the number of existing replicas (since Pangaea can perform short-cut creation), and the order in which these replicas are created (since it affects the shape of the graph). We choose an article size

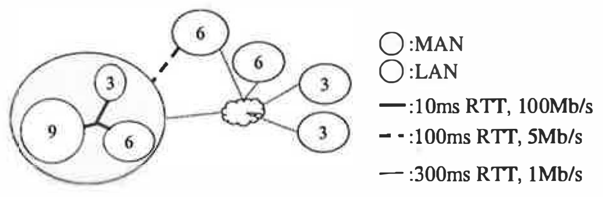


Figure 10: *Simulated network configurations modeled after our corporate network. The gray circle represents the SF bay area metropolitan-area network (MAN), the upper bubble represents Bristol (UK), and the other bubbles represent India, Israel, and Japan. The number in a circle shows the number of servers running in the LAN.*

of 50KB, a size typical in Usenet [29]. We try to average out the final parameter by creating and reading about 1000 random files for each sample point and computing the mean. We run both article posters and readers at a constant speed (≈5 articles posted or read/second), because our performance metrics are independent of request inter-arrival time.

In this benchmark, we run multiple servers in a single (physical) node to build a configuration with a realistic size. To avoid overloading the CPU or the disk, we choose to run six virtual servers on a type-B machine (Table 1), and three virtual servers on each of other machines, with the total of 36 servers on 9 physical nodes. Figure 10 shows the simulated geographical distribution of nodes, modeled after HP's corporate network. For the same logistical reasons, instead of Coda, we compare three versions of Pangaea:

pang: Pangaea with three gold replicas per new file.

hub: This configuration centralizes replica management by creating, for each file, one gold replica on a server chosen from available servers uniformly at random. Bronze replicas connect only to the gold replica. Updates can still be issued at any replica, but they are all routed through the gold replica. This roughly corresponds to Coda.

random: This configuration creates a graph by using simple random walks without considering either gold replicas or network proximity. It is chosen to test the effect of Pangaea's graph-construction policy.

We expect Pangaea's access latency to be reduced as more replicas are added, since that increases the chance of file contents being transferred to a new replica from a nearby existing replica. Figure 11 confirms this prediction. In contrast, the hub configuration shows no speedup no matter how many replicas of a file exist, because it always fetches data from the central replica.

Figure 12 shows the network bandwidth consumption during file updates. Although all the systems consume the same total amount of traffic per update (i.e.,

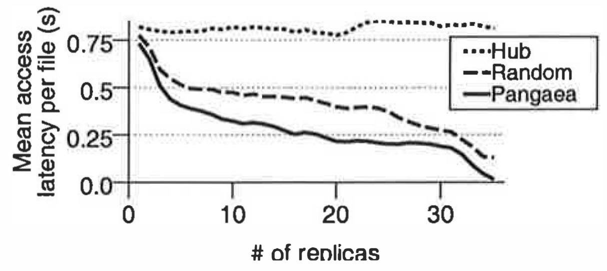


Figure 11: *The average time needed to read a new file in a collaborative environment. The X axis shows the number of existing replicas of a file. The Y axis shows the mean latency to access a file on a node that does not yet store a replica of the file.*

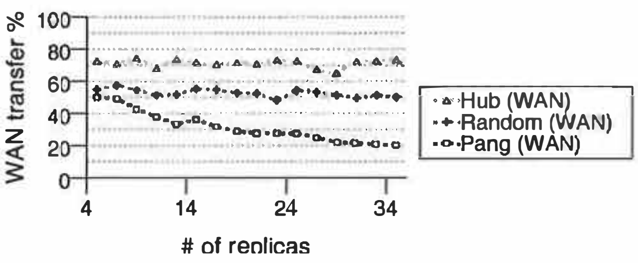


Figure 12: *Wide-area network bandwidth usage during file updates. The Y axis shows the percentage of traffic routed through the indicated networks. "WAN+MAN" shows the traffic that flowed through non-LAN (i.e., those with ≥10ms RTT), whereas "WAN" shows the traffic that flowed through networks with ≥180ms RTT (see also Figure 10).*

(#-of-replicas − 1) ∗ filesize), Pangaea uses far less wide-area network traffic since it transfers data preferentially along fast links using dynamic spanning-tree construction (Section 5.1.3). This trend becomes accentuated as more replicas are created.

Figure 13 shows the time the pang configuration took to propagate updates to replicas of files during the same experiment. The "max" lines show large fluctuations, because updates must travel over 300ms RTT links multiple times using TCP. Both numbers are independent of the number of replicas, because (given a specific network configuration) the propagation delay depends only on the graph diameter, which is three, in this configuration. We believe that 4 seconds average/15 seconds maximum delay for propagating 50KB of contents over 300ms, 1Mb/s links is reasonable. In fact, most of the time is spent in waiting when constructing a spanning tree (Section 5.1.3); cutting the delay parameter would shrink the propagation latency, but potentially would worsen the network bandwidth usage.

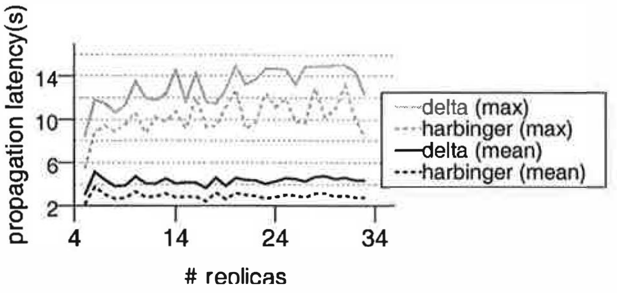


Figure 13: *The time needed to propagate updates to all replicas. The dashed lines show the time needed to distribute harbingers to replicas. They represent the window of inconsistency; i.e., time before which users may observe old contents. The solid lines represent the time needed to distribute actual updates. They represent the number of seconds users wait before seeing the new contents. The "mean" lines show the mean time needed for an update issued at one replica to arrive at all replicas, for a file with a specific number of replicas. The "max" lines show the maximum time observed for an update to arrive at all replicas of the file.*

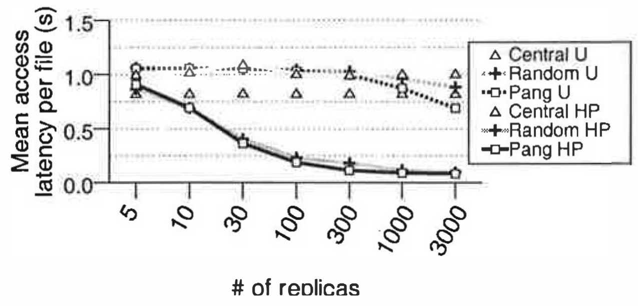


Figure 14: *File-reading latency in a simulated 3000-node system. The meaning of the numbers is the same as in Figure 11.*

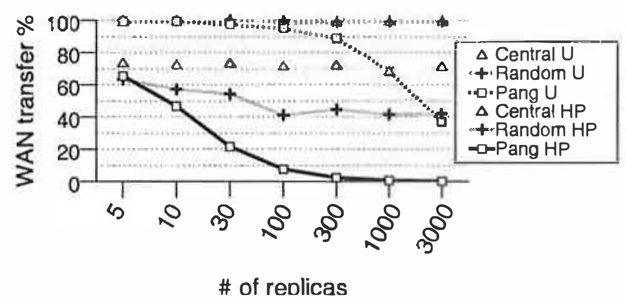


Figure 15: *Wide-area network bandwidth usage during file updates in simulated 3000-node systems. The meaning of the numbers is the same as in Figure 12.*

7.7 Performance and network economy at a large scale

The previous section demonstrated Pangaea's ability to fetch data from a nearby source and distribute updates through fast links, yet only at a small scale. This section investigates whether these benefits still hold at a truly large scale, by using a discrete event simulator that runs Pangaea's graph-maintenance and update-distribution algorithms. We extracted performance parameters from the real testbed we used in the previous section, and ran essentially the same workload as before. We test two network configurations. The first configuration, called HP, is the same as Figure 10, but the number of nodes in each LAN is increased eighty-fold, to a total of 3000 nodes. The second configuration, called U, keeps the size of each LAN at six nodes, but it increases the number of regions to 500 and connects regions using 200ms RTT, 5Mb/s links.

Figures 14 and 15 show average file-read latency and network bandwidth usage in these configurations. These figures show the same trend as before, but the differences between the configurations are more pronounced. In particular, in the HP configuration, Pangaea propagates updates almost entirely using local-area network for popular files, since it crosses over wide-area links only a fixed number of times, regardless of the number of replicas. In the U configuration, Pangaea still saves bandwidth, more visibly when many replicas exist. The systems cannot improve read latency much in U, because most of the accesses are forced to go over wide area links, but Pangaea still shows improvement with many replicas.

7.8 Availability analysis

This section studies the effects of pervasive replication, especially name-space containment, on the system's availability. A Pangaea server replicates not just replicas accessed directly by the users, but also all the intermediate directories needed to look up those replicas. Thus, we expect Pangaea to disrupt users less than traditional approaches that replicate files (or directories) on a fixed number of nodes.

We perform trace-based analysis to verify this prediction. Two types of configurations are compared: Pangaea with one to three gold replicas per file, and a system that replicates the entire file system contents on one to four nodes. Our trace was collected on our departmental file server, and it contains 24 users and 116M total accesses to 566K files [31]. To simulate a wide-area workload from this single-node trace, we assume that each user is on a different node; thus, all the simulated configurations contain 24 nodes.

For each configuration, we start from an empty file system and feed the first half of the trace to warm the system up. We then artificially introduce remote node crashes or wide-area link failures. To simulate the former situation,

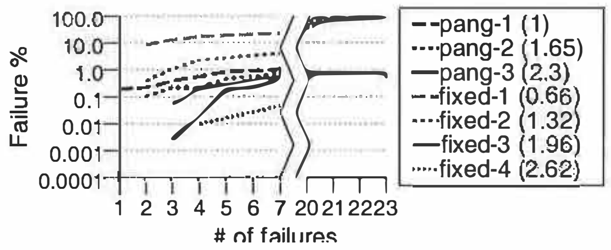


Figure 16: *Availability analysis using a file-system trace; the users of a failed node move to a functioning node. The numbers in parentheses show the overall storage consumption, normalized to* pang-1.

we crash 1 to 7 random nodes and redirect accesses by the user on a failed node to another random node. To simulate link failures, in which one to four nodes are isolated from the rest, we crash 20 to 23 random nodes and throw away future activities by the users on the crashed nodes. We then run the second half of the trace and observe how many of the users' sessions[7] can still complete successfully. We run simulation 2000 times for each configuration with different random seeds and average the results.

Figure 16 shows the results. For network partitioning, Pangaea wins by a huge margin; it shows near-100% availability thanks to pervasive replication, whereas the other configurations must rely on remote servers for much of the file operations. For node failures, the differences are smaller. However, we can still observe that for the same storage overhead, Pangaea offers better availability.

8 Conclusions

Pangaea is a wide-area file system that targets the needs for data access and sharing of distributed communities of users. It federates commodity computers provided by users. Pangaea is built on three design principles: 1) pervasive replication to provide low-access latency and high availability, 2) randomized graph-based replica management that adapts to changes in the system and conserves WAN bandwidth, and 3) optimistic consistency that allows users to access data at any time, from anywhere.

The evaluation of Pangaea shows that Pangaea is as fast and as efficient as other distributed file systems, even in a LAN. The benefits of pervasive replication and the adaptive graph-based protocols become clear in heterogeneous environments that are typical of the Internet and large intranets. In these environments, Pangaea outperforms existing systems in three aspects: access latency, efficient usage of WAN bandwidth, and file availability.

[7]We define a session to be either a directory operation (i.e., `unlink`), or a series of system calls to a file between and including `open` and `close`. If any one of the system calls fails, we consider the session to fail.

Acknowledgements

We thank our shepherd Peter Druschel, the anonymous reviewers, and the members of our group in HP Labs—especially, Eric Anderson, Mahesh Kallahalla, Kim Keeton, Susan Spence, Ram Swaminathan, and John Wilkes—for offering invaluable feedback that improved the quality of this work.

References

[1] Atul Adya, William J. Bolosky, Miguel Castro, Ronnie Chaiken, Gerald Cermak, John R. Douceur, John Howell, Jacob R. Lorch, Marvin Theimer, and Roger Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symp. on Op. Sys. Design and Impl. (OSDI)*, Boston, MA, USA, December 2002.

[2] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless Network File Systems. In *15th Symp. on Op. Sys. Principles (SOSP)*, pages 109–126, Copper Mountain, CO, USA, December 1995.

[3] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Conf. on Measurement and Modeling of Comp. Sys. (SIGMETRICS)*, pages 34–43, Santa Clara, CA, USA, June 2000.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. RFC1813: NFS version 3 protocol specification. http://www.faqs.org-/rfcs/rfc1813.html, June 1995.

[5] Pei Cao and Sandy Irani. Cost-Aware WWW proxy caching algorithms. In *1st USENIX Symp. on Internet Tech. and Sys. (USITS)*, Monterey, CA, USA, December 1997.

[6] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *18th Symp. on Op. Sys. Principles (SOSP)*, pages 202–215, Lake Louise, AB, Canada, October 2001.

[7] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. In *6th Symp. on Princ. of Distr. Comp. (PODC)*, pages 1–12, Vancouver, BC, Canada, August 1987.

[8] Armando Fox and Eric A. Brewer. Harvest, yield, and scalable tolerant systems. In *6th Workshop on Hot Topics in Operating Systems (HOTOS-VI)*, pages 174–178, Rio Rico, AZ, USA, March 1999. `http://www.csd.uch.gr/~markatos/papers/hotos.ps`.

[9] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global Internet host distance estimation service. *IEEE/ACM Trans. on Networking (TON)*, 9(5):525–540, October 2001.

[10] Richard A. Golding, Darrell D. E. Long, and John Wilkes. The refdbms distributed bibliographic database system. In *USENIX Winter Tech. Conf.*, San Francisco, CA, USA, January 1994.

[11] Jim Gray. A census of Tandem system availability between 1985 and 1990. *IEEE Trans. on Reliability*, 39(4):409–418, October 1990.

[12] Håkan Grahn and Per Stenström and Michel Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 11(3), June 1995.

[13] John Howard, Michael Kazar, Sherri Menees, David Nichols, M. Satyanarayanan, Robert Sidebotham, and Micahel West. Scale and performance in a distributed file system. *ACM Trans. on Comp. Sys. (TOCS)*, 6(1), 1988.

[14] M. Ji, E. Felten, R. Wang, and J. P. Singh. Archipelago: an island-based file system for highly available and scalable Internet services. In *USENIX Windows Systems Symposium*, August 2000.

[15] Leonard Kawell Jr., Steven Beckhart, Timoty Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. In *Conf. on Comp.-Supported Coop. Work (CSCW)*, Chapel Hill, NC, USA, October 1988.

[16] Minkyong Kim, Landon P. Cox, and Brian D. Noble. Safety, visibility, and performance in a wide-area file system. In *USENIX Conf. on File and Storage Sys. (FAST)*, Monterey, CA, January 2002. Usenix.

[17] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *9th Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS-IX)*, pages 190–201, Cambridge, MA, USA, November 2000.

[18] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *USENIX Winter Tech. Conf.*, pages 95–106, New Orleans, LA, USA, January 1995.

[19] David Mazières. A toolkit for user-level file systems. In *USENIX Annual Tech. Conf.*, Boston, MA, USA, June 2001.

[20] Lily B. Mummert, Maria R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *15th Symp. on Op. Sys. Principles (SOSP)*, pages 143–155, Copper Mountain, CO, USA, December 1995.

[21] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *18th Symp. on Op. Sys. Principles (SOSP)*, pages 174–187, Lake Louise, AB, Canada, October 2001.

[22] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *5th Symp. on Op. Sys. Design and Impl. (OSDI)*, Boston, MA, USA, December 2002.

[23] D. Scott Parker, Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce Walker, Evelyn Walton, Johanna Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. on Software Engineering*, SE-9(3):240–247, 1983.

[24] Konstantinos Psounis and Balaji Prabhakar. A randomized web-cache replacement scheme. In *Infocom*, Anchorage, AL, USA, April 2001.

[25] David H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, UC Los Angeles, 1998. Tech. Report. no. UCLA-CSD-970044.

[26] Luigi Rizzo. Dummynet, `http://info.iet.unipi.it/~luigi/ip_dummynet/`, 2001.

[27] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th Symp. on Op. Sys. Principles (SOSP)*, pages 188–201, Lake Louise, AB, Canada, October 2001.

[28] Yasushi Saito and Christos Karamanolis. Replica consistency management in the pangaea wide-area file system. Technical report, HP Labs, 2002. To be published.

[29] Yasushi Saito, Jeffrey Mogul, and Ben Verghese. A Usenet performance study, September 1998. `http://www.research.digital.com/wrl/projects/newsbench/`.

[30] Sleepycat Software. The Berkeley database, 2002. `http://sleepycat.com`.

[31] Susan Spence, Erik Riedel, and Magnus Karlsson. Adaptive consistency—patterns of sharing in a networked world. Technical Report HPL-SSP-2002-10, HP Labs, February 2002.

[32] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th Symp. on Op. Sys. Principles (SOSP)*, pages 172–183, Copper Mountain, CO, USA, December 1995.

[33] Chandramohan Thekkath, Timothy Mann, and Edward Lee. Frangipani: a scalable distributed file system. In *16th Symp. on Op. Sys. Principles (SOSP)*, pages 224–237, St. Malo, France, October 1997.

[34] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *IFIP Int. Conf. on Dist. Sys. Platforms and Open Dist. (Middleware)*, 1998. `http://www.cs.cornell.edu/Info/People/rvr/papers/pfd/pfd.ps`.

[35] Werner Vogels. File system usage in Windows NT 4.0. In *17th Symp. on Op. Sys. Principles (SOSP)*, pages 93–109, Kiawah Island, SC, USA, December 1999.

[36] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The Locus distributed operating system. In *9th Symp. on Op. Sys. Principles (SOSP)*, pages 49–70, Bretton Woods, NH, USA, October 1983.

[37] Haifeng Yu and Amin Vahdat. The Costs and Limits of Availability for Replicated Services. In *18th Symp. on Op. Sys. Principles (SOSP)*, pages 29–42, Lake Louise, AB, Canada, October 2001.

Ivy: A Read/Write Peer-to-Peer File System

Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen
{athicha, rtm, thomer, benjie}@lcs.mit.edu
MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139.

Abstract

Ivy is a multi-user read/write peer-to-peer file system. Ivy has no centralized or dedicated components, and it provides useful integrity properties without requiring users to fully trust either the underlying peer-to-peer storage system or the other users of the file system.

An Ivy file system consists solely of a set of logs, one log per participant. Ivy stores its logs in the DHash distributed hash table. Each participant finds data by consulting all logs, but performs modifications by appending only to its own log. This arrangement allows Ivy to maintain meta-data consistency without locking. Ivy users can choose which other logs to trust, an appropriate arrangement in a semi-open peer-to-peer system.

Ivy presents applications with a conventional file system interface. When the underlying network is fully connected, Ivy provides NFS-like semantics, such as close-to-open consistency. Ivy detects conflicting modifications made during a partition, and provides relevant version information to application-specific conflict resolvers. Performance measurements on a wide-area network show that Ivy is two to three times slower than NFS.

1 Introduction

This paper describes Ivy, a distributed read/write network file system. Ivy presents a single file system image that appears much like an NFS [33] file system. In contrast to NFS, Ivy does not require a dedicated server; instead, it stores all data and meta-data in the DHash [9] peer-to-peer block storage system. DHash can distribute and replicate blocks, giving Ivy the potential to be highly available. One possible application of Ivy is to support distributed projects with loosely affiliated participants.

Building a shared read-write peer-to-peer file system poses a number of challenges. First, multiple distributed writers make maintenance of consistent file system meta-data difficult. Second, unreliable participants make locking an unattractive approach for achieving meta-data consistency. Third, the participants may not fully trust each other, or may not trust that the other participants' machines have not been compromised by outsiders; thus there should be a way to ignore or un-do some or all modifications by a participant revealed to be untrustworthy. Finally, distributing file-system data over many hosts means that the system may have to cope with operation while partitioned, and may have to help applications repair conflicting updates made during a partition.

Ivy uses logs to solve the problems described above. Each participant with write access to a file system maintains a log of changes they have made to the file system. Participants scan all the logs (most recent record first) to look up file data and meta-data. Each participant maintains a private snapshot to avoid scanning all but the most recent log entries. The use of per-participant logs, instead of shared mutable data structures, allows Ivy to avoid using locks to protect meta-data. Ivy stores its logs in DHash, so a participant's logs are available even when the participant is not.

Ivy resists attacks from non-participants, and from corrupt DHash servers, by cryptographically verifying the data it retrieves from DHash. An Ivy user can cope with attacks from other Ivy users by choosing which other logs to read when looking for data, and thus which other users to trust. Ignoring a log that was once trusted might discard useful information or critical meta-data; Ivy provides tools to selectively ignore logs and to fix broken meta-data.

Ivy provides NFS-like file system semantics when the underlying network is fully connected. For example, Ivy provides close-to-open consistency. In the case of network partition, DHash replication may allow participants to modify files in multiple partitions. Ivy's logs contain version vectors that allow it to detect conflicting updates after partitions merge, and to provide version information to application-specific conflict resolvers.

The Ivy implementation uses a local NFS loop-back server [22] to provide an ordinary file system interface. Performance is within a factor of two to three of NFS. The main performance bottlenecks are network latency and the cost of generating digital signatures on data stored in DHash.

This paper makes three contributions. It describes a read/write peer-to-peer storage system; previous peer-to-peer systems have supported read-only data or data writeable by a single publisher. It describes how to design a distributed file system with useful integrity properties based on a collection of untrusted components. Finally, it explores the use of distributed hash tables as a building-block for more sophisticated systems.

Section 2 describes Ivy's design. Section 3 discusses the consistency semantics that Ivy presents to applications. Section 4 presents tools for dealing with malicious participants. Sections 5 and 6 describe Ivy's implementation and performance. Section 7 discusses related work, and Section 8 concludes.

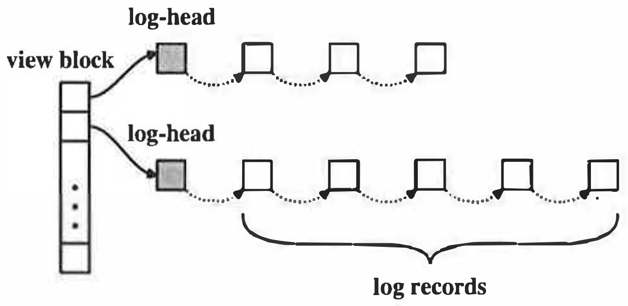


Figure 1: Example Ivy view and logs. White boxes are DHash content-hash blocks; gray boxes are public-key blocks.

2 Design

An Ivy file system consists of a set of logs, one log per participant. A log contains all of one participant's changes to file system data and meta-data. Each participant appends only to its own log, but reads from all logs. Participants store log records in the DHash distributed hash system, which provides per-record replication and authentication. Each participant maintains a mutable DHash record (called a *log-head*) that points to the participant's most recent log record. Ivy uses version vectors [27] to impose a total order on log records when reading from multiple logs. To avoid the expense of repeatedly reading the whole log, each participant maintains a private snapshot summarizing the file system state as of a recent point in time.

The Ivy implementation acts as a local loop-back NFS v3 [6] server, in cooperation with a host's in-kernel NFS client support. Consequently, Ivy presents file system semantics much like those of an NFS v3 file server.

2.1 DHash

Ivy stores all its data in DHash [9]. DHash is a distributed peer-to-peer hash table mapping keys to arbitrary values. DHash stores each key/value pair on a set of Internet hosts determined by hashing the key. This paper refers to a DHash key/value pair as a DHash block. DHash replicates blocks to avoid losing them if nodes crash.

DHash ensures the integrity of each block with one of two methods. A *content-hash block* requires the block's key to be the SHA-1 [10] cryptographic hash of the block's value; this allows anyone fetching the block to verify the value by ensuring that its SHA-1 hash matches the key. A *public-key block* requires the block's key to be a public key, and the value to be signed using the corresponding private key. DHash refuses to store a value that does not match the key. Ivy checks the authenticity of all data it retrieves from DHash. These checks prevent a malicious or buggy DHash node from forging data, limiting it to denying the existence of a block or producing a stale copy of a public-key block.

Ivy participants communicate only via DHash storage; they don't communicate directly with each other except when setting up a new file system. Ivy uses DHash content-hash blocks to store log records. Ivy stores the DHash key of a participant's most recent log record in a DHash block called the log-head; the log-head is a public-key block, so that the participant can update its value without changing its key. Each Ivy participant caches content-hash blocks locally without fear of using stale data, since content-hash blocks are immutable. An Ivy participant does not cache other participants' log-head blocks, since they may change.

Ivy uses DHash through a simple interface: `put(key, value)` and `get(key)`. Ivy assumes that, within any given network partition, DHash provides write-read consistency; that is, if `put(k, v)` completes, a subsequent `get(k)` will yield `v`. The current DHash implementation does not guarantee write-read consistency; however, techniques are known which can provide such a guarantee with high probability [19]. These techniques require that DHash replicate data and update it carefully, and might significantly decrease performance. Ivy operates best in a fully connected network, though it has support for conflict detection after operating in a partitioned network (see Section 3.4).

Ivy would in principle work with other distributed hash tables, such as PAST [32], CAN [29], Tapestry [41], or Kademlia [21].

2.2 Log Data Structure

An Ivy log consists of a linked list of immutable log records. Each log record is a DHash content-hash block. Table 1 describes fields common to all log records. The `prev` field contains the previous record's DHash key. A participant stores the DHash key of its most recent log record in its log-head block. The log-head is a public-key block with a fixed DHash key, which makes it easy

Field	Use
prev	DHash key of next oldest log record
head	DHash key of log-head
seq	per-log sequence number
timestamp	time at which record was created
version	version vector

Table 1: Fields present in all Ivy log records.

for other participants to find.

A log record contains information about a single file system modification, and corresponds roughly to an NFS operation. Table 2 describes the types of log records and the type-specific fields each contains.

Log records contain the minimum possible information to avoid unnecessary conflicts from concurrent updates by different participants. For example, a Write log record contains the newly written data, but not the file's new length or modification time. These attributes cannot be computed correctly at the time the Write record is created, since the true state of the file will only be known after all concurrent updates are known. Ivy computes that information incrementally when traversing the logs, rather than storing it explicitly as is done in UNIX i-nodes [30].

Ivy records file owners and permission modes, but does not use those attributes to enforce permissions. A user who wishes to make a file unreadable should instead encrypt the file's contents. A user should ignore the logs of people who should not be allowed to write the user's data.

Ivy identifies files and directories using 160-bit i-numbers. Log records contain the i-number(s) of the files or directories they affect. Ivy chooses i-numbers randomly to minimize the probability of multiple participants allocating the same i-number for different files. Ivy uses the 160-bit i-number as the NFS file handle.

Ivy keeps log records indefinitely, because they may be needed to help recover from a malicious participant or from a network partition.

2.3 Using the Log

For the moment, consider an Ivy file system with only one log. Ivy handles non-updating NFS requests with a single pass through the log. Requests that cause modification use one or more passes, and then append one or more records to the log. Ivy scans the log starting at the most recently appended record, pointed to by the log-head. Ivy stops scanning the log once it has gathered enough data to handle the request.

Ivy appends a record to a log as follows. First, it creates a log record containing a description of the update, typically derived from arguments in the NFS request. The new record's prev field is the DHash key of the most recent log record. Then, it inserts the new record into DHash, signs a new log-head that points to the new log record, and updates the log-head in DHash.

The following text describes how Ivy uses the log to perform selected operations.

File system creation. Ivy builds a new file system by creating a new log with an End record, an Inode record with a random i-number for the root directory, and a log-head. The user then mounts the local Ivy server as an NFS file system, using the root i-number as the NFS root file handle.

File creation. When an application creates a new file, the kernel NFS client code sends the local Ivy server an NFS CREATE request. The request contains the directory i-number and a file name. Ivy appends an Inode log record with a new random i-number and a Link record that contains the i-number, the file's name, and the directory's i-number. Ivy returns the new file's i-number in a file handle to the NFS client. If the application then writes the file, the NFS client will send a WRITE request containing the file's i-number, the written data, and the file offset; Ivy will append a Write log record containing the same information.

File name lookup. System calls such as open() that refer to file names typically generate NFS LOOKUP requests. A LOOKUP request contains a file name and a directory i-number. Ivy scans the log to find a Link record with the desired directory i-number and file name, and returns the file i-number. However, if Ivy first encounters a Unlink record that mentions the same directory i-number and name, it returns an NFS error indicating that the file does not exist.

File read. An NFS READ request contains the file's i-number, an offset within the file, and the number of bytes to read. Ivy scans the log accumulating data from Write records whose ranges overlap the range of the data to be read, while ignoring data hidden by SetAttr records that indicate file truncation.

File attributes. Some NFS requests, including GETATTR, require Ivy to include file attributes in the reply. Ivy only fully supports the file length, file modification time ("mtime"), attribute modification time ("ctime"), and link count attributes. Ivy computes these attributes incrementally as it scans the log. A file's length is determined by either the write to the highest offset since the last truncation, or by the last truncation. Mtime is determined by the timestamp in the most recent relevant log record; Ivy must return correct time attributes because NFS client cache consistency depends on it. Ivy computes the number of links to a file by counting the number of relevant Link records not canceled by Unlink and Rename records.

Type	Fields	Meaning
`Inode`	type (file, directory, or symlink), i-number, mode, owner	create new inode
`Write`	i-number, offset, data	write data to a file
`Link`	i-number, i-number of directory, name	create a directory entry
`Unlink`	i-number of directory, name	remove a file
`Rename`	i-number of directory, name, i-number of new directory, new file name	rename a file
`Prepare`	i-number of directory, file name	for exclusive operations
`Cancel`	i-number of directory, file name	for exclusive operations
`SetAttrs`	i-number, changed attributes	change file attributes
`End`	none	end of log

Table 2: Summary of Ivy log record types.

Directory listings. Ivy handles READDIR requests by accumulating all file names from relevant `Link` log records, taking more recent `Unlink` and `Rename` log records into account.

2.4 User Cooperation: Views

When multiple users write to a single Ivy file system, each source of potentially concurrent updates must have its own log; this paper refers to such sources as participants. A user who uses an Ivy file system from multiple hosts concurrently must have one log per host.

The participants in an Ivy file system agree on a *view*: the set of logs that comprise the file system. Ivy makes management of shared views convenient by providing a *view block*, a DHash content-hash block containing pointers to all log-heads in the view. A view block also contains the i-number of the root directory. A view block is immutable; if a set of users wants to form a file system with a different set of logs, they create a new view block.

A user names an Ivy file system with the content-hash key of the view block; this is essentially a self-certifying pathname [23]. Users creating a new file system must exchange public keys in advance by some out-of-band means. Once they know each other's public keys, one of them creates a view block and tells the other users the view block's DHash key.

Ivy uses the view block key to verify the view block's contents; the contents are the public keys that name and verify the participants' log-heads. A log-head contains a content-hash key that names and verifies the most recent log record. It is this reasoning that allows Ivy to verify it has retrieved correct log records from the untrusted DHash storage system. This approach requires that users exercise care when initially using a file system name; the name should come from a trusted source, or the user should inspect the view block and verify that the public keys are those of trusted users. Similarly, when a file systems' users decide to accept a new participant, they must all make a conscious decision to trust the new user and to adopt the new view block (and newly named file system). Ivy's lack of support for automatically adding new users to a view is intentional.

2.5 Combining Logs

In an Ivy file system with multiple logs, a participant's Ivy server consults all the logs to find relevant information. This means that Ivy must decide how to order the records from different logs. The order should obey causality, and all participants with the same view should choose the same order. Ivy orders the records using a version vector [27] contained in each log record.

When an Ivy participant generates a new log record, it includes two pieces of information that are later used to order the record. The `seq` field contains a numerically increasing sequence number; each log separately numbers its records from zero. The version vector field contains a tuple *U:V* for each log in the view (including the participant's own log), summarizing the participant's most recent knowledge of that log. *U* is the DHash key of the log-head of the log being described, and *V* is the DHash key of that log's most recent record. In the following discussion, a numeric *V* value refers to the sequence number contained in the record pointed to by a tuple.

Ivy orders log records by comparing the records' version vectors. For example, Ivy considers a log record with version vector *(A:5 B:7)* to be earlier in time than a record with version vector *(A:6 B:7)*: the latter vector implies that its creator had seen the record with *(A:5 B:7)*. Two version vectors *u* and *v* are *comparable* if and only if $u < v$ or $v < u$ or $u = v$. Otherwise, *u* and *v* are *concurrent*. For example, *(A:5 B:7)* and *(A:6 B:6)* are concurrent.

Simultaneous operations by different participants will result in equal or concurrent version vectors. Ivy orders equal and concurrent vectors by comparing the public keys of the two logs. If the updates affect the same file, perhaps due to a partition, the application may need to take special action to restore consistency; Section 3 ex-

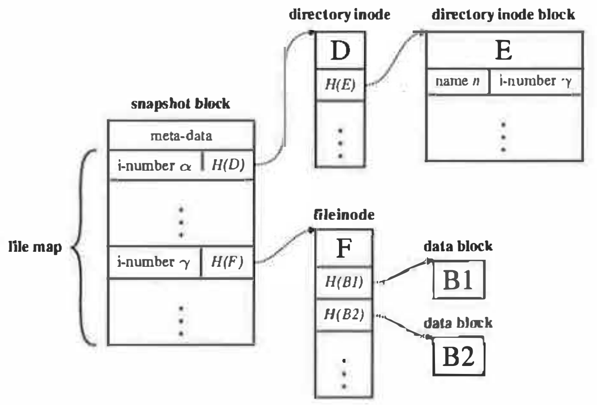


Figure 2: Snapshot data structure. $H(A)$ is the DHash content-hash of A.

plores Ivy's support for application-specific conflict resolution.

Ivy could have used a simpler method of ordering log records, such as a Lamport clock [17]. Version vectors contain more precise information than Lamport clocks about causality; Ivy uses that information to help fix conflicting updates after a partition. Version vectors help prevent a malicious participant from retroactively changing its log by pointing its log-head at a newly-constructed log; other participants' version vectors will still point to the old log's records. Finally, version vectors from one log could be used to help repair another log that has been damaged.

2.6 Snapshots

Each Ivy participant periodically constructs a private snapshot of the file system in order to avoid traversing the entire log. A snapshot contains the entire state of the file system. Participants store their snapshots in DHash to make them persistent. Each participant has its own logically private snapshot, but the fact that the different snapshots have largely identical contents means that DHash automatically shares their storage.

2.6.1 Snapshot Format

A snapshot consists of a *file map*, a set of i-nodes, and some data blocks. Each i-node is stored in its own DHash block. An i-node contains file attributes as well as a list of DHash keys of blocks holding the file's contents; in the case of a directory, the content blocks hold a list of name/i-number pairs. The file map records the DHash key of the i-node associated with each i-number. All of the blocks that make up a snapshot are content-hash blocks. Figure 2 illustrates the snapshot data structure.

2.6.2 Building Snapshots

In ordinary operation Ivy builds each new snapshot incrementally. It starts by fetching all log records (from all logs in the view) newer than the previous snapshot. It traverses these new records in temporal order. For each i-number that occurs in the new log records, Ivy maintains an i-node and a copy of the file contents. Ivy reads the initial copy of the i-node and file contents from the previous snapshot, and performs the operation indicated by each log record on this data.

After processing the new log records, Ivy writes the accumulated i-nodes and file contents to DHash. Then it computes a new file map by changing the entries corresponding to changed i-nodes and appending new entries. Ivy creates a *snapshot block* that contains the file map and the following meta-data: a pointer to the view upon which the snapshot is based, a pointer to the previous snapshot, and a version vector referring to the most recent record from each log that the snapshot incorporates. Ivy stores the snapshot block in DHash under its content-hash, and updates the participant's log-head to refer to the new snapshot.

A new user must either build a snapshot from scratch, starting from the earliest record in each log, or copy another (trusted) user's snapshot.

2.6.3 Using Snapshots

When handling an NFS request, Ivy first traverses log records newer than the snapshot; if it cannot accumulate enough information to fulfill the request, Ivy finds the missing information in the participant's latest snapshot. Ivy finds information in a snapshot based on i-number.

3 Application Semantics

This section describes the file system semantics that Ivy provides to applications, focusing primarily on the ways in which Ivy's semantics differ from those of an ordinary NFS server. Sections 3.1, 3.2, and 3.3 describe Ivy's semantics when the network provides full connectivity. Sections 3.4 and 3.5 describe what happens when the network partitions and then merges.

3.1 Cache Consistency

In general, an update operation that one Ivy participant has completed is immediately visible to operations that other participants subsequently start. The exceptions are that Ivy can't enforce this notion of consistency during network partitions (see Section 3.4), and that Ivy provides close-to-open consistency for file data (see below). Most Ivy updates are immediately visible because 1) an Ivy server performing an update waits until DHash has acknowledged receipt of the new log records and the new log-head before replying to an NFS request, and 2) Ivy asks DHash for the latest log-heads at the start of every NFS operation. Ivy caches log records, but this cache never needs to be invalidated because the records are immutable.

For file reads and writes, Ivy provides a modified form of close-to-open consistency [13]: if application A_1 writes data to a file, then closes the file, and after the close has completed another application A_2 opens the file and reads it, A_2 will see the data written by A_1. Ivy may also make written data visible before the close. Most NFS clients and servers provide this form of consistency.

Close-to-open consistency allows Ivy to avoid fetching every log-head for each NFS READ operation. Ivy caches file blocks along with the version vector at the time each block was cached. When the application opens a file and causes NFS to send an ACCESS request, Ivy fetches all the log-heads from DHash. If no other log-heads have changed since Ivy cached blocks for the file, Ivy will satisfy subsequent READ requests from cached blocks without re-fetching log-heads. While the NFS client's file data cache often satisfies READs before Ivy sees them, Ivy's cache helps when an application has written a file and then re-reads it; the NFS client can't decide whether to satisfy the reads from the cached writes since it doesn't know whether some other client has concurrently written the file, whereas Ivy can decide if that is the case by checking the other log-heads.

Ivy defers writing file data to DHash until NFS tells it that the application is closing the file. Before allowing the `close()` system call to complete, Ivy appends the written data to the log and then updates the log-head. Ivy writes the data log records to DHash in parallel to reduce latency. This arrangement allows Ivy to sign and insert a new log-head once per file close, rather than once per file write. We added a new CLOSE RPC to the NFS client to make this work. Ivy also flushes cached writes if it receives a synchronous WRITE or a COMMIT.

3.2 Concurrent Updates

Ordinary file systems have simple semantics with respect to concurrent updates: the results are as if the updates occurred one at a time in some order. These semantics are natural and relatively easy to implement in a single file server, but they are more difficult for a decentralized file system. As a result, Ivy's semantics differ slightly from those of an ordinary file server.

The simplest case is that of updates that don't affect the same data or meta-data. For example, two participants may have created new files with different names in the same directory, or might have written different bytes in the same file. In such cases Ivy ensures that both updates take effect.

If different participants simultaneously write the same bytes in the same file, the writes will likely have equal or concurrent version vectors. Recall that Ivy orders incomparable version vector by comparing the participants' public keys. When the concurrent writes have completed, all the participants will agree on their order; in this case Ivy provides the same semantics as an ordinary file system. It may be the case that the applications did not intend to generate conflicting writes; Ivy provides both tools to help applications avoid conflicts (Section 3.3) and tools to help them detect and resolve unavoidable conflicts (Section 3.4).

Serial semantics for operations that affect directory entries are harder to implement. We believe that applications rely on the file system to provide serial semantics on directory operations in order to implement locking. Ivy supports one type of locking through the use of exclusive creation of directory entries with the same name (Section 3.3). Applications that use exclusive directory creation for locking will work on Ivy.

In the following paragraphs, we discuss specific cases that Ivy differs from a centralized file system due to the lack of serialization of directory operations.

Ivy does not serialize combinations of creation and deletion of a directory entry. For example, suppose one participant calls `unlink("a")`, and a second participant calls `rename("a", "b")`. Only one of these operations can succeed. On one hand, Ivy provides the expected semantics in the sense that participants who subsequently look at the file system will agree on the order of the concurrent log records, and will thus agree on which operation succeeded. On the other hand, Ivy will return a success status to both of the two systems calls, even though only one takes effect, which would not happen in an ordinary file system.

There are cases in which an Ivy participant may read logs that are actively being updated and initially see only a subset of a set of concurrent updates. A short time later the remaining concurrent updates might appear, but be ordered before the first subset. If the updates affect the same meta-data, observers could see the file system in states that could not have occured in a serial execution. For example, suppose application A_1 executes `create("x")` and `link("x","y")`, and application A_2 on a different Ivy host concurrently executes `remove("x")`. A third application A_3 might first see just the log records from A_1, and thus see files `x` and `y`; if Ivy orders the concurrent `remove()` between the `create()` and `link()`, then A_3 might later observe that both `x` and `y` had disappeared. If the three applications compare notes they will realize that the system did not behave like a serial server.

```
ExclusiveLink(dir-inum, file, file-inum)
    append a Prepare(dir-inum, file) log record
    if file exists
        append a Cancel(dir-inum, file) record
        return EXISTS
    if another un-canceled Prepare(dir-inum, file) exists
        append a Cancel(dir-inum, file) record
        backoff()
        return ExclusiveLink(dir-inum, file, file-inum)
    append Link(dir-inum, file, file-inum) log record
    return OK
```

Figure 3: Ivy's exclusive directory entry creation algorithm.

3.3 Exclusive Create

Ordinary file system semantics require that most operations that create directory entries be exclusive. For example, trying to create a directory that already exists should fail, and creating a file that already exists should return a reference to the existing file. Ivy implements exclusive creation of directory entries because some applications use those semantics to implement locks. However, Ivy only guarantees exclusion when the network provides full connectivity.

Whenever Ivy is about to append a `Link` log record, it first ensures exclusion with a variant of two-phase commit shown in Figure 3. Ivy first appends a `Prepare` record announcing the intention to create the directory entry. This intention can be canceled by a `Cancel` record, an eventual `Link` record, or a timeout. Then, Ivy checks to see whether any other participant has appended a `Prepare` that mentions the same directory i-number and file name. If not, Ivy appends the `Link` record. If Ivy sees a different participant's `Prepare`, it appends a `Cancel` record, waits a random amount of time, and retries. If Ivy sees a different participant's `Link` record, it appends a `Cancel` record and indicates a failure.

3.4 Partitioned Updates

Ivy cannot provide the semantics outlined above if the network has partitioned. In the case of partition, Ivy's design maximizes availability at the expense of consistency, by letting updates proceed in all partitions. This approach is similar to that of Ficus [26].

Ivy is not directly aware of partitions, nor does it directly ensure that every partition has a complete copy of all the logs. Instead, Ivy depends on DHash to replicate data enough times, and in enough distinct locations, that each partition is likely to have a complete set of data. Whether this succeeds in practice depends on the sizes of the partitions, the degree of DHash replication, and the total number of DHash blocks involved in the file system. The particular case of a user intentionally disconnecting a laptop from the network could be handled by instructing the laptop's DHash server to keep replicas of all the log-heads and the user's current snapshot; there is not currently a way to ask DHash to do this.

After a partition heals, the fact that each log-head was updated from just one host prevents conflicts within individual logs; it is sufficient for the healed system to use the newest version of each log-head.

Participants in different partitions may have updated the file system in ways that conflict; this will result in concurrent version vectors. Ivy orders such version vectors following the scheme in Section 2.5, so the participants will agree on the file system contents after the partition heals.

The file system's meta-data will be internally correct after the partition heals. What this means is that if a piece of data was accessible before the partition, and neither it nor any directory leading to it was deleted in any partition, then the data will also be accessible after the partition.

However, if concurrent applications rely on file system techniques such as atomic directory creation for mutual exclusion, then applications in different partitions might update files in ways that cause the application data to be inconsistent. For example, e-mails might be appended to the same mailbox file in two partitions; after the partitions heal, this will appear as two concurrent writes to the same offset in the mailbox file. Ivy knows that the writes conflict, and automatically orders the log entries so that all participants see the same file contents after the partition heals. However, this masks the fact that some file updates are not visible, and that the user or application may have to take special steps to restore them. Ivy does not currently have an automatic mechanism for signaling such conflicts to the user; instead the user must run the `lc` tool described in the next section to discover conflicts. A better approach might be to borrow Coda's technique of making the file inaccessible until the user fixes the conflict.

3.5 Conflict Resolution

Ivy provides a tool, `lc`, that detects conflicting application updates to files; these may arise from concurrent writes to the same file by applications that are in different partitions or which do not perform appropriate locking. `lc` scans an Ivy file system's log for records with concurrent version vectors that affect the same file or directory entry. `lc` determines the point in the logs at which the partition must have occurred, and determines which participants were in which partition. `lc` then uses Ivy views to construct multiple historic views of the file system: one as of the time of partition, and one for each partition

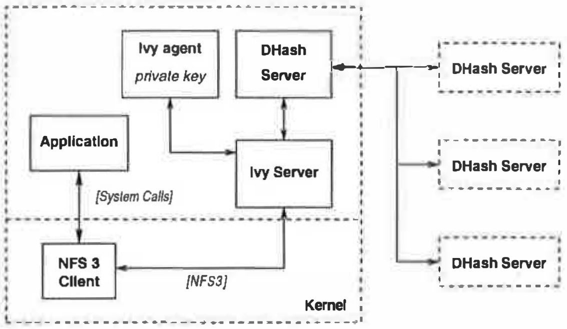


Figure 4: Ivy software structure.

just before the partition healed. For example,

```
% ./lc -v /ivy/BXz4+udjsQm4tX63UR9w71SNP0c
before: +WzW8s7fTEt6pehaB7isSfhkc68
partition1: l3qLDU5icVMRrbLvhxuJ1WkNvWs
partition2: JyCKgcsAjZ4uttbbtIX9or+qEXE
% cat /ivy/+WzW8s7fTEt6pehaB7isSfhkc68/file1
original content of file1
% cat /ivy/l3qLDU5icVMRrbLvhxuJ1WkNvWs/file1
original content of file1, changed
append on first partition
% cat /ivy/JyCKgcsAjZ4uttbbtIX9or+qEXE/file1
original content of file1
append on second partition
```

In simple cases, a user could simply examine the versions of the file and merge them by hand in a text editor. Application-specific resolvers such as those used by Coda [14, 16] could be used for more complex cases.

4 Security and Integrity

Since Ivy is intended to support distributed users with arms-length trust relationships, it must be able to recover from malicious participants. The situation we envision is that a participant's bad behavior is discovered after the fact. Malicious behavior is assumed to consist of the participant using ordinary file system operations to modify or delete data. One form of malice might be that an outsider breaks into a legitimate user's computer and modifies files stored in Ivy.

To cope with a good user turning bad, the other participants can either form a new view that excludes the bad participant's log, or form a view that only includes the log records before a certain point in time. In either case the resulting file system may be missing important meta-data. Upon user request, Ivy's `ivycheck` tool will detect and fix certain meta-data inconsistencies. `ivycheck` inspects an existing file system, finds missing `Link` and `Inode` meta-data, and creates plausible replacements in a new *fix log*. `ivycheck` can optionally look in the excluded log in order to find hints about what the missing meta-data should look like.

5 Implementation

Ivy is written in C++ and runs on FreeBSD. It uses the SFS tool-kit [22] for event-driven programming and NFS loop-back server support.

Ivy is implemented as several cooperating parts, illustrated in Figure 4. Each participating host runs an Ivy server which exposes Ivy file systems as locally-mounted NFS v3 file systems. A file system name encodes the DHash key of the file system's view block, for example, `/ivy/9RYBbWyeDVEQnxeL95LG5jJjwa4`. The Ivy server does not hold private keys; instead, each participant runs an agent to hold its private key, and the Ivy server asks the participant's local agent program to sign log heads. The Ivy server acts as a client of a local DHash server, which consults other DHash servers scattered around the network. The Ivy server also keeps a LRU cache of content-hash blocks (e.g. log records and snapshot blocks) and log-heads that it recently modified.

6 Evaluation

This section evaluates Ivy's performance 1) in a purely local configuration, 2) over a WAN, 3) as a function of the number of participants, 4) as a function of the number of DHash nodes, 5) as a function of the number of concurrent writers, and 6) as a function of the snapshot interval. The main goal of the evaluation is to understand the costs of Ivy's design in terms of network latency and cryptographic operations.

Ivy is configured to construct a snapshot every 20 new log records, or when 60 seconds have elapsed since the construction of the last snapshot. Unless otherwise stated, Ivy's block cache size is 512 blocks. DHash nodes are PlanetLab [1] nodes, running Linux 2.4.18 on 1.2 GHz Pentium III CPUs, and RON [2] nodes, running FreeBSD 4.5 on 733 MHz Pentium III CPUs. DHash was configured with replication turned off, since the replication implementation is not complete; replication would probably decrease performance significantly. Unless otherwise stated, this section reports results averaged over five runs.

The workload used to evaluate Ivy is the Modified Andrew Benchmark (MAB), which consists of five phases: (1) create a directory hierarchy, (2) copy files into these directories, (3) walk the directory hierarchy while reading attributes of each file, (4) read the files, and (5) compile the files into a program. Unless otherwise stated, the MAB and the Ivy server run on a 1.2 GHz AMD Athlon computer running FreeBSD 4.5 at MIT.

6.1 Single User MAB

Table 3 shows Ivy's performance on the phases of the MAB for a file system with just one log. All the soft-

Phase	Ivy (s)	NFS (s)
Mkdir	0.6	0.5
Create/Write	6.6	0.8
Stat	0.6	0.2
Read	1.0	0.8
Compile	10.0	5.3
Total	18.8	7.6

Table 3: Real-time in seconds to run the MAB with a single Ivy log and all software running on a single machine. The NFS column shows MAB run-time for NFS over a LAN.

Phase	Ivy (s)	NFS (s)
Mkdir	11.2	4.8
Create/Write	89.2	42.0
Stat	65.6	47.8
Read	65.8	55.6
Compile	144.2	130.2
Total	376.0	280.4

Table 4: MAB run-time with four DHash servers on a WAN. The file system contains four logs.

ware (the MAB, Ivy, and a single DHash server) ran on the same computer. To put the Ivy performance in perspective, Table 3 also shows MAB performance over NFS; the client and NFS server are connected by a 100 Mbit LAN. Note that this comparison is unfair to NFS, since NFS involved network communication while the Ivy benchmark did not.

The following analysis explains Ivy's 18.8 seconds of run-time. The MAB produces 386 NFS RPCs that modify the Ivy log. 118 of these are either MKDIR or CREATE, which require two log-head writes to achieve atomicity. 119 of the 386 RPCs are COMMITs or CLOSEs that require Ivy to flush written data to the log. Another 133 RPCs are synchronous WRITEs generated by the linker. Overall, the 386 RPCs caused Ivy to update the log-head 508 times. Computing a public-key signature uses about 14.2 milliseconds (ms) of CPU time, for a total of 7.2 seconds of CPU time.

The remaining time is spent in the Ivy server (4.9 seconds), the DHash server (2.9 seconds), and in the processes that MAB invokes (2.6 seconds). Profiling indicates that the most expensive operations in the Ivy and DHash servers are SHA-1 hashes and memory copies.

The MAB creates a total of 1.6 MBytes of file data. Ivy, in response, inserts a total of 8.8 MBytes of log and snapshot data into DHash.

6.2 Performance on a WAN

Table 4 shows the time for a single MAB instance with four DHash servers on a WAN. One DHash server runs on the same computer that is running the MAB. The average network round-trip times to the other three DHash servers are 9, 16, and 82 ms. The file system contains four logs. The benchmark only writes one of the logs, though the other three log-heads are consulted to make sure operations see the most up-to-date data. The four log-heads are stored on three DHash servers. The log-head that is being written to is stored on the DHash server with a round-trip time of 9 ms from the local machine. One log-head is stored on the server with a round-trip time of 82 ms from the local machine. The DHash servers' node IDs are chosen so that each is responsible for roughly the same number of blocks.

A typical NFS request requires Ivy to fetch the three other log-heads from DHash; this involves just one DHash network RPC per log-head. Ivy issues the three RPCs in parallel, so the time for each log-head check is governed by the largest round-trip time of 82 ms. The MAB causes Ivy to retrieve log-heads 3,346 times, for a total of 274 seconds. This latency dominates Ivy's WAN performance.

The remaining 102 seconds of MAB run-time are used in four ways. Running the MAB on a LAN takes 22 seconds, mostly in the form of CPU time. Ivy writes its log-head to DHash 508 times; each write takes 9 ms of network latency, for a total of 5 seconds. Ivy inserts 1,003 log records, some of them concurrently. The average insertion takes 54 ms (27 ms for the Chord [37] lookup, then another 27 ms for the DHash node to acknowledge receipt). This accounts for roughly 54 seconds. Finally, the local computer sends and receives 7.0 MBytes of data during the MAB run. This accounts for the remaining run time. During the experiment Ivy also inserts 358 DHash blocks while updating its snapshot; because Ivy doesn't wait for these inserts, they contribute little to the total run time.

Table 4 also shows MAB performance over wide-area NFS. The round-trip time between the NFS client and server is 79 ms, which is roughly the time it takes Ivy to fetch all the log-heads. We use NFS over UDP because it is faster for this benchmark than FreeBSD's NFS over TCP implementation. Ivy is slower than NFS because Ivy operations often require more network round-trips; for example, some NFS requests require Ivy to both fetch and update log-heads, requiring two round-trips.

6.3 Many Logs, One Writer

Figure 5 shows how Ivy's performance changes as the number of logs increases. Other than the number of logs, this experiment is identical to the one in the previous sec-

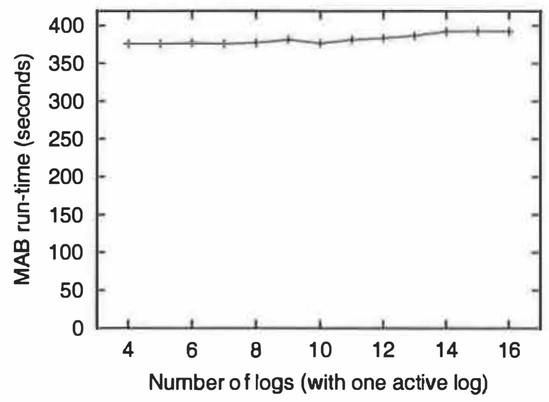


Figure 5: MAB run-time as a function of the number of logs. Only one participant is active.

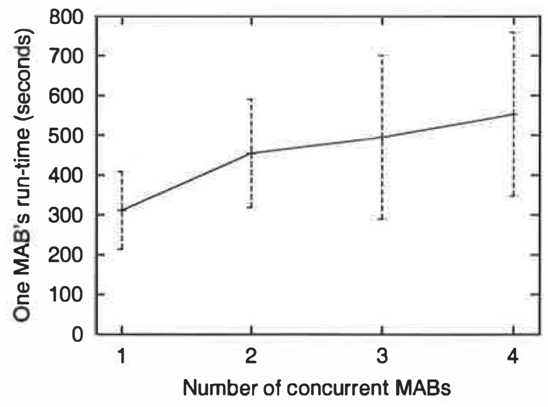


Figure 7: Average run-time of MAB when several MABs are running concurrently on different hosts on the Internet. The error bars indicate standard deviation over all the MAB runs.

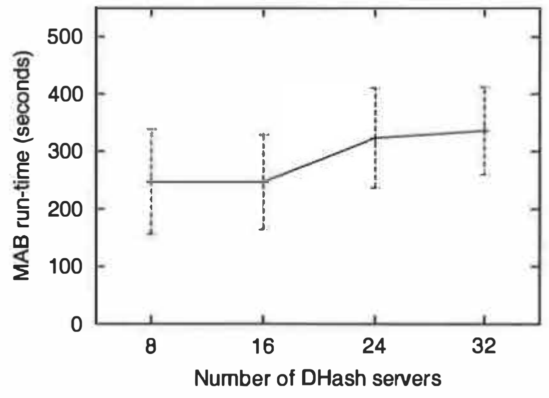


Figure 6: Average MAB run-time as the number of DHash servers increases. The error bars indicate standard deviation over different choices of PlanetLab hosts and different mappings of blocks to DHash servers.

tion. The number of logs ranges from 4 to 16, but only one participant executes the MAB — the other logs never change. Figure 5 reports results averaged over three runs.

The number of logs has relatively little impact on run-time because Ivy fetches the log-heads in parallel. There is a slight increase caused by the fact that the version vector in each log record has one 44-byte entry per participant.

6.4 Many DHash Servers

Figure 6 shows the averages and standard deviations of Ivy's MAB performance as the number of DHash servers increases from 8 to 32. For each number of servers we perform ten experimental runs. For each run, all but one of the DHash servers are placed on randomly chosen PlanetLab hosts (from a pool of 32 hosts); new log-head public keys are also used to ensure the log-heads are placed on random DHash servers. One DHash server, the Ivy server, and the MAB always execute on a host at MIT. The round-trip times from the host at MIT to the PlanetLab hosts average 32 ms, with a minimum of 1 ms, a maximum of 78 ms, and a standard deviation of 27 ms. There are four logs in total; only one of them changes.

The run-time in Figure 6 grows because more Chord messages are required to find each log record block in DHash. An average of 2.3, 2.9, 3.3, and 3.8 RPCs are required for 8, 16, 24, and 32 DHash servers, respectively. These numbers include the final DHash RPC as well as Chord lookup RPCs.

The high standard deviation in Figure 6 is due to the fact that the run-time is dominated by the round-trip times to the four particular DHash servers that store the log-heads. This means that adding more DHash servers doesn't reduce the variation.

6.5 Many Writers

Figure 7 shows the effect of multiple active writers. We perform three experiments for each number N of participants; each experiment involves one MAB running concurrently on each of N different Ivy hosts on the Internet, a file system with four logs, new log-head public keys, and 32 DHash servers. Each MAB run uses its own directory in the Ivy file system. Each data point shows the average and standard deviation of MAB run-time over the $3N$ MAB executions.

The run-time increases with the number of active participants because each has to fetch the others' newly appended log records from DHash. The run-time increases relatively slowly because Ivy fetches records from the different logs in parallel. The deviation in run-times is

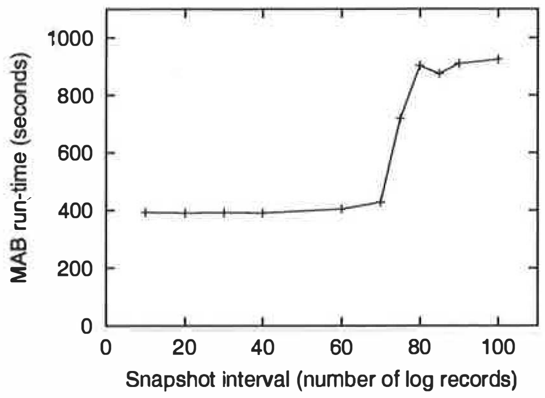


Figure 8: MAB run-time a function of the interval between snapshots. For these experiments, the size of Ivy's block cache is 80 blocks.

Phase	Ivy (s)	NFS (s)	ssh (s)
Commit	420.8	224.6	3.4
Update	284.2	135.2	2.3

Table 5: Run-times for the CVS experiment phases. DHash is running on 32 nodes on a wide-area network.

due to each participant having different network round-trip latencies to the DHash servers.

6.6 Snapshot Interval

Figure 8 shows the effect on MAB run-time of the interval between snapshots. The experiments involve one MAB instance, four logs, and 32 DHash servers. The x-axis represents the number of new log records inserted before Ivy builds each a new snapshot. For these experiments, the size of Ivy's block cache is 80 blocks. The reason the run-time increases when the interval is greater than 80 is that not all the records needed to build each snapshot can fit in the cache.

6.7 Wide-area CVS on Ivy

To evaluate Ivy's performance as a source-code or document repository, we show the run-time of some operations on a CVS [4] repository stored in Ivy. The Ivy file system has four logs stored on 32 wide-area DHash servers. The round-trip times from the Ivy host to the DHash servers storing the log-heads are 17, 36, 70, and 77 ms. The CVS repository contains 251 files and 3.3 MBytes. Before the experiment starts, two Ivy participants, X and Y, check out a copy of the repository to their local disks, and both create an Ivy snapshot of the file system. Each participant then reboots its host to ensure that no data is cached. The experiment consists of two phases. First, X commits changes to 38 files, a total of 4333 lines. Second, Y updates its local copy to reflect X's changes. Table 5 shows the run-times for the two phases. For comparison, Table 5 shows the time to perform the same CVS operations over NFS and `ssh`; in both cases the client to server round-trip latency is 77 ms.

Ivy's performance with CVS is disappointing. During a commit or update, CVS looks at every file in the repository; for each file access, Ivy checks whether some other participant has recently changed the file. CVS has locked the repository, so no such changes are possible; but Ivy doesn't know that. During a CVS commit, Ivy waits for the DHash insert of new log records and an updated log-head for each file modified; again, since CVS has locked the repository, Ivy could have written all the log records in parallel and just a single updated log-head for the whole CVS commit. A transactional interface between application and file system would help performance in this situation.

7 Related Work

Ivy was motivated by recent work on peer-to-peer storage, particularly FreeNet [8], PAST [32], and CFS [9]. The data authentication mechanisms in these systems limit them to read-only or single-publisher data, in the sense that only the original publisher of each piece of data can modify it. CFS builds a file-system on top of peer-to-peer storage, using ideas from SFSRO [11]; however, each file system is read-only. Ivy's primary contribution relative to these systems is that it uses peer-to-peer storage to build a read/write file system that multiple users can share.

7.1 Log-structured File System

Sprite LFS [31] represents a file system as a log of operations, along with a snapshot of i-number to i-node location mappings. LFS uses a single log managed by a single server in order to to speed up small write performance. Ivy uses multiple logs to let multiple participants update the file system without a central file server or lock server; Ivy does not gain any performance by use of logs.

7.2 Distributed Storage Systems

Zebra [12] maintains a per-client log of file contents, striped across multiple network nodes. Zebra serializes meta-data operations through a single meta-data server. Ivy borrows the idea of per-client logs, but extends them to meta-data as well as file contents. This allows Ivy to avoid Zebra's single meta-data server, and thus potentially achieve higher availability.

xFS [3], the Serverless Network File System, distributes both data and meta-data across participating hosts. For every piece of meta-data (e.g. an i-node) there is a host that is responsible for serializing updates to that meta-data to maintain consistency. Ivy avoids any meta-data centralization, and is therefore more suitable for wide-area use in which participants cannot be trusted to run reliable servers. However, Ivy has lower performance than xFS and adheres less strictly to serial semantics.

Frangipani [40] is a distributed file system with two layers: a distributed storage service that acts as a virtual disk and a set of symmetric file servers. Frangipani maintains fairly conventional on-disk file system structures, with small, per-server meta-data logs to improve performance and recoverability. Frangipani servers use locks to serialize updates to meta-data. This approach requires reliable and trustworthy servers.

Harp [18] uses a primary copy scheme to maintain identical replicas of the entire file system. Clients send all NFS requests to the current primary server, which serializes them. A Harp system consists of a small cluster of well managed servers, probably physically co-located. Ivy does without any central cluster of dedicated servers—at the expense of strict serial consistency.

7.3 Reclaiming Storage

The Elephant file system [34] allows all file system operations to be undone for a period defined by the user, after which the change becomes permanent. While Ivy does not currently reclaim log storage, perhaps it could adopt Elephant's version retention policies; the main obstacle is that discarding log entries would hurt Ivy's ability to recover from malicious participants. Experience with Venti [28] suggests that retaining old versions of files indefinitely may not be too expensive.

7.4 Consistency and Conflict Resolution

Coda [14, 16] allows a disconnected client to modify its own local copy of a file system, which is merged into the main replica when the client re-connects. A Coda client keeps a replay log that records modifications to the client's local copies while the client is in disconnected mode. When the client reconnects with the server, Coda propagates client's changes to the server by replaying the log on the server. Coda detects changes that conflict with changes made by other users, and presents the details of the changes to application-specific conflict resolvers. Ivy's behavior after a partition heals is similar to Coda's conflict resolution: Ivy automatically merges non-conflicting updates in the logs and lets application-specific tools handle conflicts.

Ficus [26] is a distributed file system in which any replica can be updated. Ficus automatically merges non-conflicting updates from different replicas, and uses version vectors to detect conflicting updates and to signal them to the user. Ivy also faces the problem of conflicting updates performed in different network partitions, and uses similar techniques to handle them. However, Ivy's main focus is connected operation; in this mode it provides close-to-open consistency, which Ficus does not, and (in cooperation with DHash) does a better job of automatically distributing storage over a wide-area system.

Bayou [39] represents changes to a database as a log of updates. Each update includes an application-specific *merge procedure* to resolve conflicts. Each node maintains a local log of all the updates it knows about, both its own and those by other nodes. Nodes operate primarily in a disconnected mode, and merge logs pairwise when they talk to each other. The log and the merge procedures allow a Bayou node to re-build its database after adding updates made in the past by other nodes. As updates reach a special primary node, the primary node decides the final and permanent order of log entries. Ivy differs from Bayou in a number of ways. Ivy's per-client logs allow nodes to trust each other less than they have to in Bayou. Ivy uses a distributed algorithm to order the logs, which avoids Bayou's potentially unreliable primary node. Ivy implements a single coherent data structure (the file system), rather than a database of independent entries; Ivy must ensure that updates leave the file system consistent, while Bayou shifts much of this burden to application-supplied merge procedures. Ivy's design focuses on providing serial semantics to connected clients, while Bayou focuses on managing conflicts caused by updates from disconnected clients.

7.5 Storing Data on Untrusted Servers

BFS [7], OceanStore [15], and Farsite [5] all store data on untrusted servers using Castro and Liskov's practical Byzantine agreement algorithm [7]. Multiple clients are allowed to modify a given data item; they do this by sending update operations to a small group of servers holding replicas of the data. These servers agree on which operations to apply, and in what order, using Byzantine agreement. The reason Byzantine agreement is needed is that clients cannot directly validate the data they fetch from the servers, since the data may be the result of incremental operations that no one client is aware of. In contrast, Ivy exposes the whole operation history to every client. Each Ivy client signs the head of a Merkle hash-tree [25] of its log. This allows other clients to verify that the log is correct when they retrieve it from DHash; thus Ivy clients do not need to trust the DHash servers to maintain the correctness or order of the logs. Ivy is vulnerable to DHash returning stale copies of signed log-heads; Ivy could detect stale data using techniques introduced by SUNDR [24]. Ivy's use of logs makes it slow, although this inefficiency is partially offset by its snapshot mechanism.

TDB [20], S4 [38], and PFS [36] use logging and (for TDB and PFS) collision-resistant hashes to allow modifications by malicious users or corrupted storage devices to be detected and (with S4) undone; Ivy uses similar techniques in a distributed file system context.

Spreitzer et al. [35] suggest ways to use cryptographically signed log entries to prevent servers from tampering with client updates or producing inconsistent log orderings; this is in the context of Bayou-like systems. Ivy's logs are simpler than Bayou's, since only one client writes any given log. This allows Ivy to protect log integrity, despite untrusted DHash servers, by relatively simple per-client use of cryptographic hashes and public key signatures.

8 Conclusion

This paper presents Ivy, a multi-user read/write peer-to-peer file system. Ivy is suitable for small groups of cooperating participants who do not have (or do not want) a single central server. Ivy can operate in a relatively open peer-to-peer environment because it does not require participants to trust each other.

An Ivy file system consists solely of a set of logs, one log per participant. This arrangement avoids the need for locking to maintain integrity of Ivy meta-data. Participants periodically take snapshots of the file system to minimize time spent reading the logs. Use of per-participant logs allows Ivy users to choose which other participants to trust.

Due to its decentralized design, Ivy provides slightly non-traditional file system semantics; concurrent updates can generate conflicting log records. Ivy provides several tools to automate conflict resolution. More work is under way to improve them.

Experimental results show that the Ivy prototype is two to three times slower than NFS. Ivy is available from `http://www.pdos.lcs.mit.edu/ivy/`.

Acknowledgments

We thank M. Satyanarayanan and Carnegie-Mellon University for making the Modified Andrew Benchmark available. We are grateful to the PlanetLab and RON testbeds for letting us run wide-area experiments. Sameer Ajmani, Trevor Blackwell, Miguel Castro, Josh Cates, Russ Cox, Peter Druschel, Frans Kaashoek, Alex Lewin, David Mazières, and Rodrigo Rodrigues gave us helpful feedback about the design and description of Ivy.

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933, and by grants from NTT Corporation under the NTT-MIT collaboration.

References

[1] PlanetLab. `http://www.planet-lab.org/`.

[2] D. Andersen, H. Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proc. of the ACM Symposium on Operating System Principles*, pages 109–126, December 1995.

[4] B. Berliner. CVS II: Parallelizing software development. In *Proc. Winter 1990 USENIX Technical Conference*, 1990.

[5] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *ACM SIGMETRICS Conference*, June 2000.

[6] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

[8] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.

[9] F. Dabek, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[10] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, April 1995.

[11] K. Fu, M. Frans Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pages 181–196, October 2000.

[12] J. Hartman and J. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[13] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[14] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proc. of the ACM Symposium on Operating System Principles*, pages 213–225, 1991.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. of ACM ASPLOS*, pages 190–201, November 2000.

[16] P. Kumar and M. Satyanarayanan. Log-based directory resolution in the Coda file system. In *Proc. of the Second International Conference on Parallel and Distributed Information Systems*, pages 202–213, January 1993.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[18] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proc. of the ACM Symposium on Operating System Principles*, pages 226–38, 1991.

[19] N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in content addressable networks. In *Proc. of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[20] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pages 135–150, October 2000.

[21] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[22] D. Mazières. A toolkit for user-level file systems. In *Proc. of the Usenix Technical Conference*, pages 261–274, June 2001.

[23] D. Mazières, M. Kaminsky, M. Frans Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. of the ACM Symposium on Operating System Principles*, December 1999.

[24] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, 2002.

[25] R. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer-Verlag, 1987.

[26] T. Page, R. Guy, G. Popek, and J. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report UCLA-CSD 910005, 1991.

[27] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. In *IEEE Transactions on Software Engineering*, volume 9(3), pages 240–247, 1983.

[28] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proc. of the Conference on File and Storage Technologies (FAST)*, January 2002.

[29] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, pages 161–172, August 2001.

[30] D. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[31] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[32] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[33] R. Sandberg, D. Goldberg, D. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proc. Usenix Summer Conference*, pages 119–130, June 1985.

[34] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proc. of the ACM Symposium on Operating System Principles*, pages 110–123, 1999.

[35] M. Spreitzer, M. Theimer, K. Petersen, A. Demers, and D. Terry. Dealing with server corruption in weakly consistent, replicated data systems. In *Proc. of the ACM/IEEE MobiCom Conference*, September 1997.

[36] C. Stein, J. Howard, and M. Seltzer. Unifying file system protection. In *Proc. of the USENIX Technical Conference*, pages 79–90, 2001.

[37] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM*, August 2001.

[38] J. Strunk, G. Goodson, M. Scheinholtz, and C. Soules. Self-securing storage: Protecting data in compromised systems. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pages 165–179, October 2000.

[39] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the ACM Symposium on Operating System Principles*, pages 172–183, December 1995.

[40] C. Thekkath, T. Mann, and E. Lee. Frangipani: A scalable distributed file system. In *Proc. of the ACM Symposium on Operating System Principles*, pages 224–237, 1997.

[41] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001.

Defensive Programming: Using an Annotation Toolkit to Build DoS-Resistant Software

Xiaohu Qie, Ruoming Pang, and Larry Peterson
Department of Computer Science, Princeton University
{qiexh,rpang,llp}@cs.princeton.edu

Abstract

This paper describes a toolkit to help improve the robustness of code against DoS attacks. We observe that when developing software, programmers primarily focus on functionality. Protecting code from attacks is often considered the responsibility of the OS, firewalls and intrusion detection systems. As a result, many DoS vulnerabilities are not discovered until the system is attacked and the damage is done. Instead of reacting to attacks after the fact, this paper argues that a better solution is to make software *defensive* by systematically injecting protection mechanisms into the code itself. Our toolkit provides an API that programmers use to annotate their code. At runtime, these annotations serve as both sensors and actuators: watching for resource abuse and taking the appropriate action should abuse be detected. This paper presents the design and implementation of the toolkit, as well as evaluation of its effectiveness with three widely-deployed network services.

1 Introduction

Denial-of-Service (DoS) attacks are a major source of concern in the Internet. Unlike security break-ins that obtain privileged access, DoS attacks are designed to consume a disproportionate amount of resources on the target system by exploiting weakness in the network software. When successful, such attacks make the system unavailable to well-behaved users.

Common defenses against DoS attacks include using firewalls and Intrusion Detection Systems (IDS) to monitor network links for offending traffic, as well as applying software patches to fix known vulnerabilities. However, such defensive practices burden the system administrator with making sure all systems have the up-to-date patches installed and all firewalls are properly configured. To make matters worse, even after a new attack is recognized, it is not until the vulnerabilities exploited by the attack are determined that a patch can be developed.

We observe that many DoS vulnerabilities can be attributed to the separation of software functionality and protection. When developing software, programmers primarily focus on functionality. Protection from attacks is often considered the responsibility of the OS, firewalls, and IDS, and thus not an immediate concern. As a result, many vulnerabilities in the code are not discovered until the system is hit by an attack that exploits the weakness, that is, after the damage is done.

Instead of reacting to attacks, we propose a new approach to DoS protection: *defensive programming*, by which we mean programmers embed *general* mechanisms into their software to provide *systematic* and *proactive* protection against DoS attacks. Ideally, defensive software guarantees availability even under a previously unknown DoS attack. An important aspect of this approach is that it should be designed to thwart common DoS attack characteristics; programmers should *not* have to scan their code for a specific implementation vulnerabilities and fix them, as they do when writing a software patch.

Towards this end, this paper describes our experience developing mechanisms to help programmers systematically build robust software. The key idea is to insert annotations that monitor and control the execution of the program at runtime. These annotations serve both as sensors that detect anomalies and actuators that change the control flow of a program when they detect that defensive measures are necessary. The advantage of annotations is that they allow us to adjust the program's behavior at a very fine granularity, thereby making it possible to confine the damage of an attack without negatively affecting other aspects of the program.

We have developed a toolkit consisting of a set of annotation primitives, a runtime library, and a set of compiler extensions. As a means of specifying a resource management policy, a programmer inserts annotation primitives into code so that the annotations mark where resources are acquired/released/consumed, where the program branches into independent functionalities, and what principals are holding resources. The compiler extensions check consistency among annotations by analyzing the control flow graph of the program and generating necessary code to be executed at annotated points. At runtime, appropriate monitor and control functions are invoked as control flow passes through these annotations.

The toolkit helps programmers reason about DoS problems in a more structured way. Rather than focus on implementation details, they are asked to identify the services provided and the resources consumed by their program at a high level. For example, if the programmer annotates a certain function as performing an identifiable service, the toolkit will confine a DoS attack on that service to requests of the same service, rather than letting the attack bring down the program as a whole. The flip-side, of course, is that the toolkit is not a panacea. Like any mechanism, the effectiveness of the toolkit depends on whether a good defensive policy can be specified, which is ultimately the responsibility of the programmer.

The paper makes two contributions. First, it studies the general question of how to develop defensive code that protects itself from DoS attacks. In the process, the paper identifies a class of attacks that exploits a vulnerability existing in many network servers, but that has not received attention in the literature. Second, it describes a specific mechanism—the annotation toolkit—that evolved from this study. We have implemented the toolkit in Linux, and demonstrated how to annotate widely deployed software, including the Linux IP protocol stack, the Flash web server [10], and the Linux NIS servers [8]. Our experience shows that we can significantly improve the robustness of software against DoS attacks with relatively low programming effort.

2 Related Work

Our approach to writing defensive code draws on previous research in several areas. This section explains how our work fits in this larger design space.

2.1 Intrusion Detection Systems

Anomaly detection uses statistics of normal behavior as a baseline, and treats changes in these patterns as an indication of an attack. Researchers have demonstrated that examining the sequence of system-calls made by an application is a viable approach to detecting security violations due to bugs in the program (mainly buffer overflows) [6, 14, 18].

However, current anomaly detection techniques have difficulty detecting resource-exhausting attacks, because a DoS attacker can request the same service as a legitimate user. Our approach has the flavor of anomaly detection, but with a focus on resource usage rather than security. Since the target of a DoS attack is some resource on the victim system, we instrument the program to look for irregularities in resource usage and actively participate in resource management. In a way, we do not have to distinguish DoS attacks from other activities, the rationale being that as long as resources are properly managed, the damage any DoS attack can cause is limited.

2.2 Performance Monitoring

Resource-exhausting DoS attacks often cause performance degradation on the target, making it possible to detect such attacks by monitoring the profiling data. In general, however, we found profiling-based detection insufficient for the following reasons. First, profiling does not cover all the important aspects of a program's behavior. The target resource of a DoS attack is not necessarily CPU cycles; sometimes it can be application-level objects. Second, getting the average behavior from profiling data is not enough because even perfectly legitimate users can deviate significantly from the average without attacking the system. To infer the behavior distribution from profiling data is a hard problem that does not have a good solution for the general case.

In order to collect comprehensive data for analysis and extract meaningful information from the data, it is necessary to know what resources a program consumes, as well as where and how they are being used. Our annotation interface allows a program to provide such information.

In performance assertion checking [11], the original program is instrumented to generate an execution log, which is then checked offline for performance violations. The assertions and logging facilities are the counter parts of our resource sensors. Being independent of the original program, assertions can be declared in a more expressive language. In contrast, our resource sensors and actuators are part of the original program being annotated, monitoring resource usage and changing the program control flow at runtime.

In addition, our goal is not only detection, but also protection. Since an appropriate defensive action is highly dependent on the functionality and architecture of the program, the action has to be specified at the source code level. Watching profiling data can sometimes tell us the system is being attacked, but without a defense mechanism built into the program, the only available response is to kill the victim process, which is a DoS attack in its own right.

2.3 Static Code Analysis

There has recently been much work in automatic detection of software errors and security bugs through static code analysis. Recent work by Engler *et al.* [4, 5] introduced the technique of *meta-level compilation*. The idea is that the software must obey certain rules for correctness, such as "kernel code cannot call blocking functions with interrupts disabled" and "message handlers must free their buffer before completing". System programmers specify the rules in a high-level language, and an extensible compiler then applies the rules throughout the program source to check for violations. Meta-level compilation is very successful in finding errors in OS code, as well as a wide range of security bugs using rules such as "do not dereference user pointers without checking validity". The authors found several DoS possibilities in the kernel code they examined, but the result is limited to a special case in which an attacker controls the iterations of a kernel loop.

Static analysis alone is not sufficient for detecting DoS attacks since such attacks do not necessarily rely on software bugs. It is often the cumulative pressure on resources that puts a system in peril, even though the software itself is bug-free. Thus, besides examining how the software is implemented, we must also watch how it is *executed*. Such information can be only collected at runtime with additional application or OS support. Previous work in detecting race conditions in concurrent programs [12] seems to support this point of view. Our approach differs from previous work of static analysis mainly in that we check for possible "rule" violations at runtime, with a focus on resource usage.

2.4 OS Mechanisms

There has been an ongoing effort to build new OS mechanisms and specialized OSs to provide service differentiation and guarantees. For example, Resource Containers [3] are an abstraction that takes over the process' role as the primary resource principal. It allows multiple cooperating processes to bind to the same container, as well as a process to change its resource and schedule binding dynamically when it executes on behalf of another activity. The Scout operating system [9, 15] uses a similar abstraction—the path—as the primary resource and schedule principal. Both systems have been shown to be able to defend against certain flooding DoS attacks. The improvement results from more accurate resource accounting and service isolation.

An important contribution of resource containers is the separation of resource principals and execution domains, but as an OS approach, resource management policies are ultimately enforced via process scheduling among execution domains. In case an execution domain multiplexes among a set of resource principals, resource containers reduce to a passive accounting facility. However, many functionality-rich services, such as web servers and routing daemons, are single-process-event-driven. *Intra-process* protection is more important for these applications, since we do not want to penalize the entire process when just one of its functions is being abused. This calls for a finer-grain resource protection than what can be provided by an OS approach. Using annotations inserted by the programmer to monitor and control the execution path within a process, our approach offers a finer-grain protection than OS approaches.

Another system-level approach, SEDA [19], proposes a programming model in which a program is divided into stages and each stage enforces its own resource management policy by controlling threads running in that stage. This model differs from the traditional process-based resource protection in that resource allocation not only depends on the process, but also on the stage in which the process is running. From this perspective, our approach is similar to SEDA. On the other hand, SEDA is not intended for DoS protection, and does not protect resources that cannot be protected by scheduling.

Finally, our toolkit is intended to improve the robustness of existing software. Annotating code is more programmer-friendly than imposing a new OS architecture or abstraction, which often requires re-architecting code. This is especially true with Scout and SEDA.

3 DoS Attack Characterization

Researchers have studied many DoS attacks [13, 7]. What is lacking, however, is an analysis of their common characteristics: what they attack and how they attack it. Such a characterization would help us understand the signature of DoS attacks, and shed light on how to systematically and proactively write defensive software.

There are several well-known attacks on network software, including the ICMP flood attack (send a large number of ICMP echo packets at the target), TCP SYN attack (flood the target with connection-open requests), and Christmas Tree packets (overwhelm a target with packets that have exceptional bits turned on in the header---e.g., IP options—dictating the packet receive special processing). A less well-known attack, which we refer to as route cache poisoning, involves an attacker flooding a router with packets carrying a sequence of nonsensical IP addresses (e.g., "1", "2", "3", and so on), thereby blowing the router's first level route cache. This causes the router's control processor to spend all its time building new microcode and loading it into the switch engine. This happens at the expense of the router responding to its neighbors' routing probes, which causes the neighbors to believe the router is down.

These examples illustrate that DoS attacks abuse a legitimate service by sending it a large volume of requests, suggesting that rate limiting [17] and load conditioning [19] would be an effective defense. However, DoS attacks can also be carried out in a way that renders rate limiting strategies ineffective. The following example illustrates this possibility.

3.1 Slow TCP Attacks

Many TCP-based services follow the request-reply paradigm. Since a server must set aside resources while a client request is being processed, it is possible to exhaust the server's resource by manipulating the operation of TCP. The idea behind the attack is for the client to make the TCP connection as slow as possible. This simple idea can be realized in three different ways.

First, a client can send the request very slowly. Since

TCP is a byte-stream protocol without record boundaries, the server cannot interpret the client's request until all the data is received. Suppose a request contains 2000 bytes, and the TCP MSS is 1000 bytes. Under normal operation, the client would send the request in two packets. If, instead, the client sends the request one byte at a time, which does not violate any protocol and application requirements, it would take 2000 RTTs before the server can start to process the request. The client can insert additional delays between packets to further extend the duration.

Second, once the server starts to send results back, the client can read the data very slowly. The server side TCP would interpret the closed TCP advertised window in the acknowledgment packet as a signal that the client application is temporarily busy, thus pause sending.[1] The server will not be able to send more data until the window is opened again. Thus by abusing TCP's flow control mechanism the client can pace the rate of data sent by the server.

Third, the client can acknowledge the response very slowly by pretending the packet was lost. Without seeing an acknowledgment, the server will retransmit. Similar to the slow receiver, the client can pace the sending rate of the server by controlling when to acknowledge a packet. In this scenario, the client abuses TCP's reliable transmission feature.

One target of the Slow TCP attack is web servers. Being a slow sender, an attacker can construct an extremely long HTTP request (e.g., copy the header "`User-Agent: Slow TCP Sender \r\n`" 5000 times) and send it at a very low rate (e.g. 1 byte every 50 seconds). Being a slow receiver or ACKer, an attacker just requests a big file then nibbles at the server's output. The goal of the attacker is to keep the connection alive as long as possible. Since the number of concurrent connections a web server can maintain is limited, given sufficient number of slow attackers, the server's available connections will be exhausted, and all subsequent requests will be denied.

We verified this idea experimentally by implementing a HTTP request generator that uses slow TCP, and tested it against two popular web servers: Apache [2] and Flash [10]. The attack proves to be extremely effective. Despite the fact that TCP has a keep-alive timer, the Linux TCP implementation limits the number of retransmission attempts to 12, and both Apache and Flash have built-in mechanisms to time-out idle connections, all three forms of slow attacks are able tie up a connection for several days, causing the servers to disappear from the net. We were also able to attack NIS servers in a similar way.

In general, we believe such attacks are not limited to TCP servers. For example, an attacker could disable a firewall that provides NAT or Proxy services by repetitively sending packets from all available ports to a random set of destinations. Once the translation table on the firewall is filled up, other users are effectively cut off from the rest of the Internet.

[1]After some time, the server TCP will send a 1-byte packet to test if the client has consumed any data.

3.2 Attacks Revisited

When characterizing DoS attacks, it is helpful to distinguish between two types of resources: *renewable resources*, such as CPU cycles, the bandwidth of network, disks, and buses; and *non-renewable resources*, such as processes, ports, buffers, PCBs, and locks. To attack a renewable resource, the attacker continually consumes the resource so that legitimate services do not receive enough of the resource over time. This is usually achieved by flooding the server with massive number of requests in order to keep the target system busy. In contrast, if the target resource is non-renewable, the attacker tries to acquire as many resource as possible and does not release them. This form of attack does not require flooding to make the target busy.

In the rest of the paper we denote an attack targeting a renewable resource a *busy* attack, and an attack targeting a non-renewable resource a *claim-and-hold* attack. However, we note that some attacks cannot be clearly placed in one category. For instance, the target resource of SYN flooding attack is half-open connections, which is a non-renewable resource, but to exhaust this particular resource, the attacker must keep the system busy with a flood of new requests. In another example, router cache poisoning succeeds when the router's CPU is overwhelmed, thus it is a busy attack, yet it works by directly attacking the route cache, which is a non-renewable resource.

These "exceptions" are not special cases, but in fact, phenomenon due to the duality between busy and claim-and-hold attacks. Often in mending one vulnerability, we open the system to another vulnerability. For example, the Apache web server sets a limit of 150 connections to protect itself from runaway resource consumption, yet by enforcing this limit, connections become a "scarce" resource and the program is potentially vulnerable to claim-and-hold attacks. On the other hand, to protect non-renewable resources, the system must perform a recycling function when the resource becomes unavailable. This function itself could become an accessory in a busy attack if it is not resource-controlled. This is the weakness exploited by the route cache poisoning attack. Clearly, a general defense mechanism must protect the system from both types of vulnerabilities at the same time; watching only one type of attacks is not sufficient.

4 Defensive Strategies

Our overall strategy is to separate resources among activities in a program along two dimensions. For renewable resources, we balance resource usage among program functionalities, thereby confining the impact of an attack to the individual service being attacked. For non-renewable resources, we identify principals that hold non-renewable resources and reclaim resources from principals that are not making minimal progress. These two aspects of our strategy are discussed in turn.

4.1 Busy Attack Defense

The strategy is to balance resource usage among program functionalities, thereby confining the impact of an attack to the individual service being attacked. Towards this end, we introduce the concept of *service* and propose a resource control mechanism with actuators at service entries and sensors at resource access points.

4.1.1 Services and Resources

We define a *service* to be a program component that provides an *independent* functionality. Each service, in turn, consumes some amount of renewable resources. Figure 1 shows the conceptual model of a server program divided into services. Client requests are served by different services, as they execute a code path through the program, and multiple services share various resources.

There is often a clear correspondence between services and program code paths, and in many cases, a service is implemented by a particular function and associated subroutines. For example, in the Linux kernel, each ICMP service is handled by a distinct function with name `icmp_<service>` (e.g. `icmp_echo`). Thus, a program can be divided into services according to code paths. To expose the service structure of a program, we ask programmers to annotate the service entry functions in their programs. We have also built a set of compiler tools to help user check coverage and consistency of service annotations.

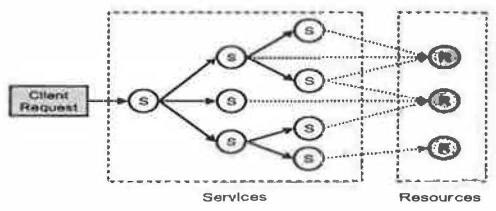


Figure 1: Service View of a Program

We assume each service is performed by a function. When this is not the case, the programmer must extract the part of code that performs the service and wrap it in a separate function. Our experience with the Flash web server and the Linux TCP/IP code suggests that there are few places we need to do the extraction and all of them are straightforward. The benefit of marking functions instead of arbitrary code regions as services is that the user need only annotate service entry points. Our compiler can then automatically annotate the corresponding service exit points, thereby reduce the overall programmer workload and the chance of inconsistent marking. Also, the service hierarchy structure is clearly represented by the function call graph.

Services can be disjoint or nested. For example, in the Linux IP stack (Figure 2), `TCP-recv` and `UDP-recv` are disjoint services, while the service of `IP options` processing is nested inside IP processing. Nested services allow the programmer to divide a coarse-grain service into finer-grain sub-services. Dividing services in this way has the advantage of confining the damage of an attack within a smaller range. When a nested service tries to over-use some resource, action is taken only on the inner-most service that directly uses the resource, for fear that doing anything to the parent services may over-penalize sibling services. For example, if we further divide the service of IP option handling into a sub-service for every type of IP option, then when the code dealing with one type of option is attacked, all other IP options can be still be handled normally.

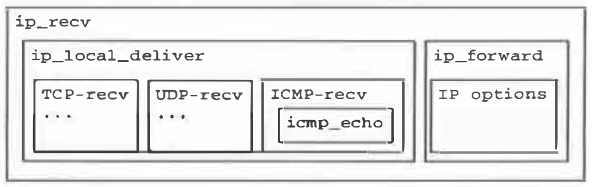


Figure 2: Services in Linux IP Stack

As services correspond to code paths, we can control resource usage of a service by rate-limiting execution on its code paths, especially the "expensive" ones. For example, the Linux kernel checks a rate limit when deciding whether to send out an ICMP packet. We can view the act of changing from one execution path to another, based on resource usage, as intra-process "scheduling" among services. However, since we do not know which code path will be attacked, and it is hard to precisely tell how expensive code paths are, there are two interesting questions in rate-limiting code paths: 1) where to place sensors that monitor resource usage and actuators that change the program execution path; and 2) at what rates code paths should be limited, or how to decide whether or not to switch out of the current code path each time execution reaches the actuators.

4.1.2 Sensors and Actuators

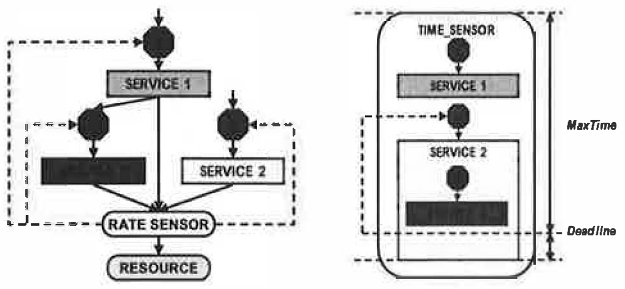
We need a systematic way to place sensors and actuators in the program, because placing them in an ad hoc way may leave holes to be exploited—the code path being attacked might not have an annotation on it. On the other hand, we want to minimize the number of annotations, especially actuators, because switching out of a code path needs to be handled in a program-specific way, and it takes programmer's effort to write such a handler.

Rate limiters found in existing software, such as the Linux kernel, are actually a composite component that consists of both a sensor and an actuator: the sensor monitors the execution rate on the code path, and the actuator deflects the execution to another code path when the rate limit is violated. This approach works well because we know which potential attacks we want to defend against and therefore can put rate limiters on the right code path. In our case, we do not assume that we know about any particular attack. With this different assumption, we found that actuators and sensors need to be placed at different locations in the program, in order that (1) actuation happens at the right place, and (2) resource usages to be properly limited. The following discusses the placement of actuators and sensors, in turn.

For actuators that control the execution path, we argue that service entry points are the right place for them to be placed. This is for three reasons. First, a service is the unit of fault isolation, and activities within the same service share fate. Therefore, it is better to not begin processing a service request if it cannot acquire enough resources to complete. Second, it is easier to abort or delay processing a service request at the entry point than in the midst of processing. Third, each service needs only one actuator, thus the total number of actuators depends only on the number of services.

A potential trade-off here is that sometimes at the service entrance we may not be able to precisely predict whether a request can get enough resource. However, in all busy attacks we know of, a service must be invoked at a high rate in order to exhaust system resource. Therefore, the effect of this inaccuracy is minor, because it matters only when the service is about to reach its resource quota. In other words, rate precision is not so important in DoS defense, as we are not making QoS guarantees. Finally, the programmer may define finer-grained services to achieve better precision.

For sensors that monitors resource usage, we argue that they should be put at resource access points, for example, where a system call is invoked to transmit a packet. If we were to put sensors together with actuators at service entries, it would require much effort and experience to set an appropriate rate limit for each service because it is unclear how service rate limits would map onto actual resource usage. As we try to set the limits for services before knowing which service will be attacked, there is a risk of being either too conservative or too optimistic. Also, the choice is often host-specific



(a) Rate Control (b) Time Control

Figure 3: Managing Renewable Resources

and cannot be easily shared or reused. In contrast, it is straightforward to measure the resource usage at the point the resource is accessed, and it is relatively easy for the programmer to specify an overall rate limit for each type of resource.

Taken together, the sensors monitor both the overall resource usage and usages by individual services, thereby affecting admission decisions at actuators placed at service entry points. (See Figure 3(a)). Actuators control admission to any service that tries to consume disproportional amount of resource. Further details about the actual mechanism is discussed in Section 5.3.

4.1.3 Controlling Continuous Resource

The discussion to this point assumes that resources are always consumed at particular locations of the program. We further distinguish between two types of renewable resources: *discrete* resources, which include almost all renewable resource except CPU time (e.g. network/disk bandwidth); and *continuous* resources, which include CPU time. Unlike discrete resources, CPU time is spent continuously as the program executes, so we can no longer monitor resource on some particular code paths. Therefore it needs to be managed differently.

There are mainly two questions: how to detect CPU overload and how to locate the service being exploited. Our approach is to ask the user to specify time limits on some high-level functions for each invocation, and we control admission to the downstream service that violates the deadline, as shown in Figure 3(b). Again, more details are given in Section 5.3.

4.2 Claim-and-Hold Attack Defense

In order to consume renewable resources, the attacking activity must be *active*, i.e., executing code on the CPU. This observation has greatly simplified our solution to defend busy attacks—basically we need to control the execution frequency and duration of different code paths. Protecting non-renewable resources,

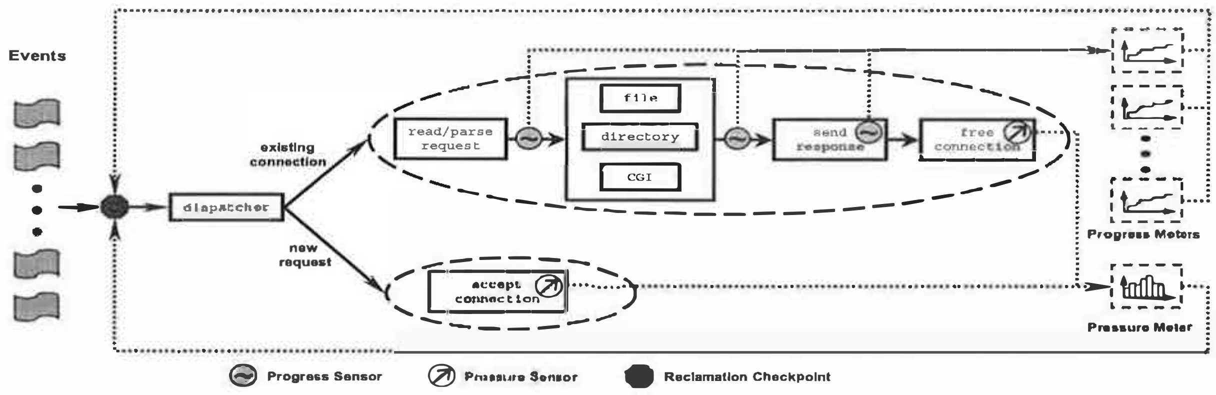


Figure 4: Managing Non-renewable Resources in an Event-driven Web Server

however, is a different story. Attackers holding the resource do not necessarily have to remain active once the resource is acquired.

Protecting non-renewable resources is essentially a process of specifying a replacement policy: when the resource becomes exhausted, which ones should be reclaimed. Resources can be reclaimed either periodically or when some event indicates recycling is necessary. Thus, the problem boils down to one of deciding: (1) what resources to reclaim, and (2) when to reclaim them. We introduce two metrics—*progress* and *pressure*—that characterize these two aspects of a replacement policy, respectively. Our defense strategy involves annotating a program with sensors and actuators that set and react to these two metrics.

4.2.1 Progress and Pressure

In the Slow TCP attack against web servers, the resource in question is the server connection. Neither Apache or Flash implements an explicit replacement policy. When connections are exhausted, the server simply rejects new requests. The connection resource is returned when the client request is completed. It is also reclaimed by timers. In Flash, there are two build time parameters, CGI_TIMELIMIT and IDLEC_TIMELIMIT. The former caps the maximum running time of a CGI program forked by a client request, and the latter controls the maximum period a client can be idle. When either limit is exceeded, the connection is dropped and resources associated with this connection are freed.

The weakness of this simple mechanism lies in the fact that an attacker can trick the server into thinking it is still in the middle of a request, thereby holding resource without triggering the timers. Alternatively, to guarantee availability, we could choose to tear down the oldest connection when the connection table becomes full. The problem with this approach is that it is biased against clients on a slow link or those downloading a large file.

A better solution is to measure how well a client is making use of the resources it has acquired, and combine this information with other metrics such as age. A client should be allowed to hold resources longer than others, as long as it has a good reason. We use *progress* to denote such a metric. The exact form of progress depends on the resource and application in question, but in general, a proper progress metric should reflect how the principal holding a resource is making use of it. Progress is expected to increase proportionally with time. In the web server example, how many bytes the server has sent to the client could be used to construct the progress metric.

A replacement policy also has to specify when to reclaim resources. Since recycling itself could be an expensive operation, uncontrolled invocations also open up the possibility of busy attacks, which is what we saw in the route cache poisoning attack. We define a *pressure* metric to control the invocation of the reclaim function. Intuitively, resources should be recycled when the pressure on it exceeds a certain threshold, which could be caused either by too many clients requesting the resource, or no clients releasing the resource.

Programmers can develop other metrics tailored to the application. As a general toolkit, we currently only support interfaces to keep track of progress and pressure, on top of which a variety of policies can be built.

4.2.2 Placing Sensors and Actuators

Figure 4 illustrates how sensors and actuators can be placed in an event-driven web server such as Flash.

Sensors are inserted into a program to track (record) progress in two ways. If the principal in question generates output of some kind, the unit of the output is a natural measure of progress; e.g., one can annotate a program with a progress sensor that records how many bytes have been read or written, how many packets have been forwarded, and so on. In a second scenario, an entire task can be broken into stages, where progress is recorded when the task moves from one stage to the next. For example, the Flash web server breaks client request processing into three stages: request reading and parsing, back-end processing, and result sending. Some stages can be further divided depending on the operations required by a particular request (e.g, requesting a static page vs. dynamic content). A stage is represented by a unique "handler" associated with a connection. In this example, progress sensors can be placed where the connection handler is changed.

It is usually obvious how to insert sensors into a program to track pressure: there are often well-defined points in the program where non-renewable resources are accessed; e.g., inside resource allocators and deallocators. Pressure sensors can be placed at these points. Some abstract non-renewable resources are not accessed via an explicit function interface, in which case we need the programmer to annotate the points at which the resource is acquired and released.

Turning to the actuator side, there is a single reclamation actuator that is a function of both metrics: it decides to reclaim resources if the pressure metric is greater than some threshold, and should this be the case, it uses the progress metric to decide which instance of the resource to reclaim. Reclamation actuators are placed in two types of locations. First, the trigger role of the pressure sensor suggests that a reclamation actuator should be placed immediately after a pressure sensor. In fact, we we envision a combined pressure annotation marking the point where resources are claimed and released.

In addition, however, pressure also needs to be examined periodically, as it could build up even in the absence of activity. This implies that we also need to insert a reclamation actuator—which we call a *reclamation checkpoint* to distinguish it from the combined pressure sensor/actuator—that is periodically visited by the control flow. For most server programs this is not a problem as they are iterative by nature. For example in Flash, we could place such an actuator inside its main event loop, as shown in Figure 4. An important issue however, is that when an action is taken, it must not leave the server in an inconsistent state; e.g., not free all resources associated with an activity, or continue to reference a principal that is no longer valid due to the reclamation. We do not have a general solution to the problem, except that by imposing transaction semantics the risk of inconsistency can be reduced. In other words, the checkpoint should be placed outside all functions that are considered atomic.

Finally, when placing a reclamation checkpoint we need to consider how often it is visited by the program control flow. If the interval is not properly bounded, we effectively lose control on the resource. One way to preserve granularity is to use the techniques presented in the previous section, such as the time-sensor, to limit the branches leaving the checkpoint. But under extreme situations, for instance an attacker causing the program to enter an infinite loop, we could still lose control. We considered other alternatives, such as using a timer signal to perform resource checking, but it is extremely hard to perform resource reclamation in a signal handler while still guaranteeing such operations do not lead to inconsistencies. We consider this as one limitation of intra-process protection—sometimes we need to depend on inter-process protection provided by the OS. In other words, there is a trade-off between absolute control and preserving the original program structure.

5 Annotation Toolkit

This section describes our annotation toolkit in detail, focusing first on the annotations themselves, and then on the underlying implementation.

5.1 Renewable Resource Management

The toolkit includes annotations that are used to denote admission control upon service entry, plus annotations that serve as sensors for monitoring rate and time limits. We consider each in turn.

• SERVICE_ADMISSION(min_rate)

The user marks a function as a service entry point, specifying the minimum rate at which that service is allowed to proceed. For example, the following is from the service that satisfies cold cache requests in the Flash web server:

```
SRCode
ProcessColdRequest(httpd_conn* hc)
{
  if (!SERVICE_ADMISSION(3))
    return SR_PLEASE_TRY_AGAIN_LATER;
  /* rest of the function ... */
}
```

This annotation does not directly change the execution path of the program, but returns a hint on whether the service should be admitted based on its resource usage, allowing the program to (1) do necessary cleanup before aborting, (2) delay servicing the request, or (3) ignore the hint. The annotation takes parameter *min_rate* and always returns 1 when the service is invoked below the minimal rate, regardless whether the service has used up its resource quota. This allows users to guarantee service rate for some important services under resource contention.

• RATE_SENSOR (max_rate, weight)

This annotation is used to specify the maximal weighted rate for a particular code path. For example, in order to rate-limit the packet and byte rates of ICMP, we may annotate the code with the following lines before ICMP pushes a packet to IP:

```
  if (!RATE_SENSOR (sysctl_icmp_max_msg_rate, 1))
    icmp_msg_rate_violation++;
  if (!RATE_SENSOR (sysctl_icmp_max_byte_rate, msg_size))
    icmp_byte_rate_violation++;
  ip_build_xmit(...);
```

RATE_SENSOR can be placed any where in the program, unlike SERVICE_ADMISSION which must be put at function entries. It returns a hint on whether the current measured rate of the code path is within the specified maximal rate. However, it is completely legitimate for programmer to ignore the hint (as in the example above) if the limit is not strict. This is because the annotation sends feed-back to the service admission point, thereby eventually limiting resource usage to the specified rate.

• TIME_SENSOR (max_time)

This annotation is used to monitor the execution time of a function (and its subroutines) on each invocation. It is applied on functions in the same way as SERVICE_ADMISSION. For example, to control the execution time of an event handler in Flash web server, we extract the invocation of the event handler into a separate function and annotate the function with TIME_SENSOR so that admission to services invoked by event handlers will be bounded by the time limit.

```
static void LaunchHandler(...)
{
  TIME_SENSOR(handlerTimeLimit);
  handler(tempConn, i, do_what);
}
```

5.2 Non-renewable Resource Management

The toolkit also includes a set of annotations that both demark the allocation and freeing of non-renewable resources, and check to see if resources need to be reclaimed.

• RESOURCE_DECL(resid)

This annotation declares a non-renewable resource that needs protection, where *resid* is a unique identifier. The annotation initializes a data structure to represent the resource. This annotation should be placed in the initialization part of a program.

• RESOURCE_ACQUIRED(resid, p, amt)
• RESOURCE_RELEASED(resid, p, amt)

These two annotations take an opaque pointer and the amount of resource being accessed. The pointer serves to identify the principal; it is usually an application-specific data structure. The annotation also records the timestamp of the operation in order to calculate the duration of resource being held by the principal.

• PRESSURE_SENSOR(resid, s)

This annotation records pressure on the resource caused by discrete events, such as a new request being denied due to the lack of resources. The second argument can be used to express the severity of the situation.

• RESOURCE_UNAVAILABLE(resid)
• RESOURCE_AVAILABLE(resid)

Some applications disable new requests as soon as the resource is used. In this scenario, pressure cannot be tracked in a discrete fashion. Instead, pressure accumulates continually over time when no resources are released. These two annotations are used in such situations.

• PROGRESS_SENSOR(resid, p, prog)

This annotation updates the progress metric of a principal by *prog*. The use of the opaque pointer *p* should be consistent with that in RESOURCE_ACQUIRED and RESOURCE_RELEASED.

• RECLAMATION_CHECKPOINT(resid, cb, min_pres, min_prog)

This annotation is the actuator that performs resource recycling. By default, it takes resources back from the principal making the least progress. Programmers can configure the operation with two additional parameters: *min_pres* specifies that actions should be taken only when the pressure exceeds certain threshold; *min_prog* restricts the actions to be taken only upon principals making less progress than the parameter. By setting different thresholds, a programmer can control the frequency of recycling and give principals that have already made significant progress an allowance to finish the task. Programmers also need to specify a callback function *cb* that is invoked by the actuator. It should free resources associated with a principal (identified by the opaque pointer), but can also be used to log activity for offline analysis.

5.3 Implementation Details

Each annotation is implemented as a C-macro, and is linked with an instance of a corresponding data structure. Key data structures in our toolkit include service, rate sensor, time sensor, resource, and principal, with each maintaining a different set of counters.

A service structure contains a rate counter for service entry rate so that it can tell whether the entry rate is below the minimal rate given in the annotation. It also contains flags to indicate resource or time limit violation by the service. The rate counter is reset to zero at the end of every period (a period lasts for one second in our prototype). The violation flag is also adjusted periodically.

To account resource usage of services, global variable *current_service* points to the service currently being executed. As services can be nested, the variable is updated on each service entry and exit. (Our compiler extension inserts service exit calls corresponding to SERVICE_ADMISSION annotations.) The following gives pseudo-code for service admission and exit:

```
do_service_admission (svc_id, min_rate) {
  if (at the end of period)
    adjust rate and time violation;
  update service entry counter;
  check_deadline();
  set current_service to svc_id;
  if (service within min_rate || there is no violation)
    return 1;
  return 0;
}

do_service_exit () {
  check_deadline();
  set current_service to parent service;
}
```

The rate sensor structure contains a rate counter *for each service* that uses the rate sensor and a counter for the overall rate. In addition, it maintains a shared rate limit for services: whenever a rate counter of any service exceeds the shared rate limit, the service is marked with a rate-limit violation flag, and its subsequent admissions will be rejected until the end of the period (with the exception of services that are admitted because they are below the minimal service rate). The shared rate limit is adjusted at the end of each period with *additive increase / multiplicative decrease (AIMD)* depending on whether the overall rate exceeds the given limit on the sensor. Below is the pseudo-code for rate sensor:

```
do_rate_sensor(rate_id, max_rate) {
  if (at the end of period)
    adjust shared limit AIMD (total rate counter, max_rate);
  update per service and total rate counters;
  if (per service counter > shared limit) {
    set rate violation on current_service;
    return 0;
  }
  return (rate_counter(rate_id) <= max_rate);
}
```

Adjusting the shared rate limit dynamically allows more flexible rate control than computing the limit with min-max algorithm, which assumes that every service obeys the shared limit. The programmer may allow some service to use more resources than the common share—by overriding it with minimal service rate or ignoring the result of SERVICE_ADMISSION —but the shared rate limit is adjusted to a level so that the overall rate still matches the specified limit. This allows users to make application-specific decision on resource allocation other than purely "fair" sharing.

Like the SERVICE_ADMISSION annotation, the scope of a TIME_SENSOR annotation includes the current function and all its subroutines. At entry TIME_SENSOR computes and stores a deadline in global variable *current_deadline*. When TIME_SENSOR is applied in a user-space process, the time-stamp is obtained by getting process usage time (which is process time plus system time on behalf on the process) in order to exclude the impact of process scheduling. (In contrast, SERVICE_ADMISSION and RATE_SENSOR uses wall time.) Within the scope of time-limit, the current time is compared against *current_deadline* (see the pseudo-code for check_time_limit below) at each service entry and exit. If the deadline is missed, the current service is marked as the violating service and following services will not check the deadline any more. The service being marked as the violating service will be rejected admission for some penalty period (with the same exception of minimal service rate), at which time violation flag on the service is reset to 0. The duration of the penalty period depends on by how much time the service violates the time limit.

```
do_time_sensor(max_time) {
  current_deadline = current_usage_time + max_time;
  passed_deadline = 0;
}

check_deadline() {
  if(!passed_deadline && current_usage_time > current_deadline) {
    time_violation(current_service) +=
      penalty(current_usage_time - current_deadline);
    passed_deadline = 1;
  }
}
```

The implementation of the interface for non-renewable resource management is straight-forward. Most macros simply update the pressure or progress counter in the data structure representing a resource or a principal. As an example, we give pseudo-code for RECLAMATION_CHECKPOINT:

```
do_reclamation_checkpoint(resid, cb, min_pres, min_prog) {
  update pressure on resid;
  if (pressure(resid) > min_pres) {
    for (each pri holding the resource) {
      usage(pri) += (time_now - last_timestamp) * held_amt(pri);
      normalized_prog(pri) = absolute_prog(pri) / usage(pri);
      update worst_pri by comparing normalized_prog counters;
    }
    /* worst_pri records the pri making the least progress */
    if (normalized_prog(worst_pri) < min_prog)
      (*cb)(worst_pri);
  }
}
```

The only trick in the code is that comparisons are made in *normalized* progress, rather than *absolute* progress, as reported directly by the application via the PROGRESS_SENSOR macro. The reason is that comparing absolute progress is not fair to young principals that have not yet received enough time to make progress. Intuitively, a principal holding resources for a longer period of time should have made better progress.

5.4 Compiler Support

Because code path annotations are tightly coupled with program control flow structure, we instrumented GCC and built some small tools to help users annotate their code. In general, the compiler automatically adds auxiliary annotations to complete those marked by user, and links the code annotation with the toolkit data structures. It also checks consistency of annotations and gives warning on potential discrepencies.

GCC builds a syntax tree for each function body after parsing. We added our extension to a hook between parsing and intermediate language (RTL) generation. The compiler extension traverses syntax trees to look for service admission/time sensor annotations and function exit points. When a function is marked with a service admission/time sensor annotation, the compiler inserts a call to the corresponding service exit/time sensor exit functions before each function exit.

The instrumented GCC also writes the control flow graph to a file. Our code path analyzer then reads this file and gives warnings for following cases: (1) there is a path from an entry function to a rate-sensor annotation that does not go through any service admission annotation, and (2) there are some expensive operations (e.g. loops and library function calls) enclosed by a time-sensor annotation and not enclosed by any service admission annotation.

6 Evaluation

We experimentally tested our toolkit on widely deployed software: the Flash web server, Linux kernel networking code, and NIS (yellow page) server. For each example, we annotate the code by asking ourselves the same set of questions—what services need to be separated and what resources need protection. We then tested the robustness of both the unmodified and annotated servers under various attacks. We found that both busy and claim-and-hold attack vulnerabilities exist in all test cases, and that by exploiting these vulnerabilities, an attacker could either disable, or seriously degrade the level of service. The annotated servers are much more resilient under the attacks, which demonstrates the generality and effectiveness of our toolkit. We also found situations where our toolkit has difficulty in providing protection to the desirable level. We identify some as implementation issues that can be improved by extending our toolkit, while others are fundamental limitations of our approach.

6.1 Flash Web Server

6.1.1 Annotating Flash Web Server

Flash [10] is a web server with a single-process-event-driven architecture. The main loop launches connection handlers on I/O events. We first annotate every handler function called in main loop as a service entry point. Since some of these handlers implement more than one *independent* functions—e.g., it may either read a file or execute a CGI program—we mark nested services in top-level services by functionality (e.g., `CGIStuff`). There are also some functions that contain loops or make system calls (and thus have potential to be attacked). One such example is `MakeCrossedString`, which concatenates parts of a cross-buffer string. Such functions are also marked as separate services for fault isolation. A fourth class of functions perform non-critical tasks—e.g., `ReduceCacheIfNeeded`—which we also mark as services. Altogether, 46 services are annotated.

To limit time spent in each event handler function invocation, we extract the handler function call in main loop and place it in a separate function, called `LaunchHandler`, and annotate this function with `TIME_SENSOR` .

All non-renewable resources in Flash are consumed on behalf of a connection, which is itself a non-renewable resource. Flash disables new requests when *numConnects* reaches the upper limit. The following code illustrates how we annotated function `AcceptConnections`—we insert two sensors to track usage and pressure on the connection resource. Note the pointer to the `http_conn` data structure is used as the principal identifier.

```
int AcceptConnections(int cnum, int acceptMany) {
  httpd_conn* c;
  do {
    PrepareConnOnAccept(c, newConnFD, &sin);
    numConnects++;
    RESOURCE_ACQUIRED(HTTPCONN, c, 1);
  } while (numConnects < maxConnects && acceptMany);
  if (numConnects >= maxConnects) {
    DisallowNewClients();
    RESOURCE_UNAVAILABLE(HTTPCONN);
  }
}
```

A typical HTTP connection goes through three phases: request reading and parsing, back-end processing (fetch a file from disk or execute a CGI program), and result sending. A connection makes progress when it moves to the next phase or sends out bytes. Thus, progress sensors are inserted where the "state" of a connection changes and data is sent out: `DoConnReadingBackend` and `DoSingleReadBackend` are two examples of functions with embedded progress sensors.

```
DoConnReadingBackend(httpd_conn* c, int fd, int doReqReading)
{
  switch(ProcessRequestReading(c)) {
    case PRR_DONE:
      /* end of request reading */
      PROGRESS_SENSOR(HTTPCONN, c, 10000);
      break;  /* switch connection to the next phase */
    ...
  }
}
```

```
DoSingleWriteBackend(httpd_conn* c, int fd, int testing)
{
  sz = writev(c->hc_fd, ioBufs, numIOBufs);
  ...
  /* Ok, we wrote something. */
  PROGRESS_SENSOR(HTTPCONN, c, sz);
}
```

Finally, we explicitly declare the connection resource before entering the server loop and insert a checkpoint inside the loop. The annotated main loop is shown below. `DoneWithConnection` is a Flash-provided resource deallocator, here conveniently used as the callback function for connection recycling. The choice of the parameters *min_pres* and *min_prog* are explained in Section 6.1.3.

```
void MainLoop(void) {
  RESOURCE_DECL(HTTPCONN);
  for (;;) {
    RECLAMATION_CHECKPOINT(HTTPCONN, DoneWithConnection, 5, 500);
    for (each I/O event) {
      Launchhandler(handler, tempConn, ...);
    }
    if (!newClientsDisallowed) AcceptConnections(-1, TRUE);
  }
}
```

6.1.2 Slash Attack

Flash is a very robust program: disk operations and CGI jobs are separated into helper processes rather than performed by the main process, thereby allowing the OS to protect the main process. Flash also has some built-in mechanisms to control its resource consumption; e.g. calls to `fork()` are already rate-limited. However, it is extremely difficult to write a program free of vulnerabilities, and Flash is not an exception. We found the following code in function `ExpandSymlinks`, which parses a "cold" URL that is not in server's hot URL cache:

```
/* Remove any leading slashes. */
while ( rest[0] == '/' )
  {
    (void) strcpy( rest, &(rest[1]) );
    --restlen;
  }
```

The loop has time complexity quadratic in number of leading slashes. As Flash does not limit the length of a URL, a URL with many leading slashes takes a lot time to parse: it takes 150 ms on a PIII 700 machine to remove $10,000$ leading slahes from a URL; 7 such requests per second is enough to saturate an un-annotated server.

Our attacker is a simple program that sends HTTP request "GET /////.../*id*" to the Flash server, where $id = 1, 2, 3, \ldots$ to avoid duplicate URLs. Under attack, the un-annotated server soon reaches the maximum number of connections. Subsequent connection requests enter a connection queue waiting to be accepted. The server will accept a connection every 150 ms. Thus server response time is greater than the connection request queue length $\times$ 150 ms.

Slash attack serves our purpose well because it shows that implementation inefficiencies that lead to DoS vulnerability may appear at unexpected locations in the source code. Ad hoc protection is not likely to cover such a vulnerability and we need a systematic approach for DoS defense. *Importantly, we knew about this problem to formulate the attack, but we did not need to have knowledge of this bug when annotating the code.*

For a Flash server that is annotated with service admissions and a time sensor with a limit of 20 ms on `LaunchHandler`, the attack has no effect on requests of hot URLs. The annotations recognizes that service `ProcessColdRequestBackend2` (see Figure 5) takes too much time on each invocation and rate limits the

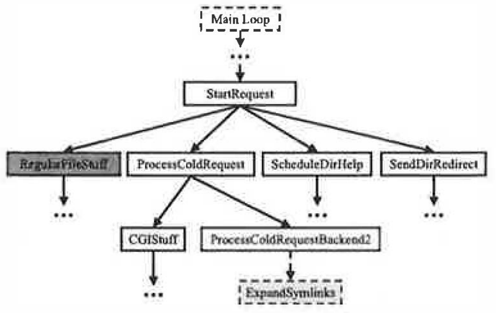


Figure 5: Position of ExpandSymlinks in Flash service hierarchy

service depending on how much time it takes for each invocation. The connection is closed on service admission rejection, so that connections do not accumulate over time. Service `ProcessColdRequestBackend2` is not invoked for "hot" URLs. By limiting CPU spent for cold URLs, we insure access to hot pages under slash attack.

no attacker	4.3 ms
attacker #slash = 0	4.3 ms
attacker #slash = 10000, original	25,000 ms
attacker #slash = 10000, annotated	5.1 ms

Table 1: Flash response time under slash attack

Table 1 compares the average response time for a "hot" 10KB file for both original Flash and annotated Flash, when the server is under slash attack. The slash attacker sends 10 requests per second to saturate the Flash server. We first measure response time to a single client without any attacker present. We then measure response time to client when there is a competing client; i.e. the attacker sends ten requests per second but with no leading slashes in the URL. The third row shows response time from an unprotected Flash server under attack, a 5000$\times$ slow down. The last row shows the response time from an annotated Flash server. The small increase of response time for annotated Flash under attack is because Flash processes a cold URL periodically and thus delays the hot request for up to 150 ms. Despite this small fluctuation, the response time from an annotated Flash server does not change by much on average under slash attack.

On the other hand, access to cold URLs is limited for annotated Flash under slash attack. The probability of success for a cold request is linear to the ratio between the user request rate and the attack request rate. For example, if an attacker sends ten requests per second (which is enough to saturate an unprotected server) and the user sends one request per second, then with probability 50% it takes no more than than $\log 0.5 / \log 0.9 = 6.57$ requests to access a cold

URL. However, since nothing prevents the attacker from sending requests at a higher rate, clients may not be able to access "cold" pages in many attempts. This phenomenon shows that the effectiveness of fault isolation depends on service granularity, and sometimes depends on program classification granularity. If Flash were to further classify requests into ones with short URLs and those with long URLs, the impact of a slash attack would be further limited.

6.1.3 Slow TCP Attack

In unmodified Flash, the connection resource is recycled by an idle timer associated with each connection. The default time-out value `IDLEC_TIMELIMIT` is 500 seconds. The timer is reset by any event on the socket, such as data arrival or TCP send buffer becoming available. Thus, to launch a successful claim-and-hold attack, an attacker needs to generate an event before the 500 second timer expires. Once the available connections run out, the unmodified Flash server enters the "denial-of-service" mode, disallowing new clients. Our Slow TCP based clients can easily cause the situation to persist for days without generating very much network traffic.

By comparison, the annotated Flash server is able to recover from the "denial-of-service" mode by recycling connections. Our current toolkit implementation uses a sliding window to record pressure history. Setting *min_pres* to 5 instructs the server to reclaim resources from unproductive connections after it has been disallowing new clients for about 5 seconds. The progress of each client is tracked as follows: when a connection moves from one stage to another the absolute progress of the connection is incremented by a numerical value of 10000; when the connection is in the final result sending stage, its absolute progress increases as the bytes being successfully written. In conjunction with the *min_prog* of 500, the server enforces the following policy: a client should not stay in one stage (other than the last one) for more than 20 seconds, otherwise its normalized progress will drop below $10000/20 = 500$ and be considered "unproductive". Once in the final stage, the client should read at least 500 bytes of the server's response per second. With these resource limits, well-behaved clients including those on slow links go largely unaffected, but claim-and-hold attackers are no longer able to tie up server resources for unreasonably long periods of time.

Note that by specifying a single progress-and-pressure threshold, we may not be able to completely eliminate the vulnerability to Slow TCP attacks. Attackers can still open many connections and make each request proceed slowly while staying just above the acceptable progress threshold. To solve this problem, the programmer can specify a more refined defensive policy with the toolkit: for example, under resource pressure, at most one third of the connections can be "very slow", another one third can be "slow", while the rest have to be "fast" connections. This can be accomplished by putting more than one checkpoint with multi-level progress-and-pressure thresholds, so that the server will recycle resources more aggressively under higher pressure.

6.1.4 Overhead

Regarding programming overhead, we add in total 57 annotations into Flash source, which has more than 12,000 lines of code. 46 of the annotations are service admission primitives that divide the program into fine-grain services. The rest are annotations on individual kinds of resources; e.g., CPU time and HTTP connections. As the annotations specify general resource policies, they should be able to defend against not only the attacks in the experiments, but also other potential attacks targeting the annotated services and resources.

In terms of request response time or server bandwidth we did not observe any performance degradation caused by annotation in our measurements. Table 2 reports the number of annotation primitives invoked on a typical HTTP request and the general cost of each annotation. The number of annotations executed varies depending on the file's size and whether it is in server cache, which affects the call graph, the number of server iterations, and the number of outgoing packets. The cost of each annotation is given in the number of instructions and "timestamp" operations. The exact cost of timestamp depends on whether the code being annotated is in kernel or user-space.

Primitives	Invocations per HTTP connection	Instructions/ timestamps per call
SERVICE Entry/Exit	$13 - 31$	63/2
RATE_SENSOR	*n/a*	25/1
TIME_SENSOR	*iterations*	36/2
RESOURCE_ACQUIRED	1	62/1
RESOURCE_RELEASED	1	42/0
PROGRESS_SENSOR	$2 + pkts$	23/0
RECLAMATION_CHECKPOINT	*iterations*	121/1 per principal

Table 2: Annotation Overhead

Note the 121 instructions are the worst-case cost of `RECLAMATION_CHECKPOINT` when the pressure is high and each connection is checked. Also not shown in the table is certain background processing of the toolkit library, which executes once per second for each annotation and contains less than 20 instructions per invocation.

6.2 Linux Networking Code

6.2.1 Annotating Linux Network Code

We annotate part of Linux 2.4 network code to protect network outgoing bandwidth. Our goal is to insure that no single network activity can monopolize outgoing network bandwidth. (For incoming network bandwidth, protection on local host may not be enough, however, we may want to limit CPU time spent on incoming packets for hosts with high-bandwidth network connections.)

Initially, we mark service entry points at the "send message" function of each protocol; e.g. udp_sendmsg. This gives us protocol isolation. However, icmp_reply is an interesting case since it is called by multiple functions for sending different types of ICMP messages, e.g. icmp_echo and icmp_timestamp. To have fault isolation between different types of ICMP messages, we push the service entry at icmp_reply into functions for every type of ICMP message that calls icmp_reply. For example, icmp_echo is now a service entry function, while icmp_reply is no longer marked as a service. icmp_send presents another interesting case: it is called at 13 locations to report different network errors. To prevent one type of error from suppressing others, we wrap each call site as a service. In total, we mark 27 services.

Since we may not be able to get notification about delivery of packets for protocols like ICMP, we cannot apply congestion control to manage bandwidth, as the Congestion Manager does [1]. Instead, we simply rate-limit messages from all protocols except TCP.[2] On code paths that call ip_build_xmit, we insert a call to ip_rate_control, which includes RATE_SENSOR annotations:

```
static __inline__ int ip_rate_control(int msg_size)
{
  int res = 1;
  if (!RATE_SENSOR (sysctl_ip_max_msg_rate, 1)) {
    res = 0; ip_msg_rate_violation++;
  }
  if (!RATE_SENSOR (sysctl_ip_max_byte_rate, msg_size)) {
    res = 0; ip_byte_rate_violation++;
  }
  return res;
}
```

The user can adjust sysctl_ip_max_msg_rate and sysctl_ip_max_byte_rate through the /proc file system.

6.2.2 ICMP-Echo Flood Attack

To simulate ICMP-echo flood attack, the attacker sends a flood of ICMP-echo packets to the victim using the 'ping -f' command. The attack has a 100Mbps network link and the victim is on a 10Mbps link. The victim also runs a Flash web server so that we can measure how it is affected by the attack.

Without protection, access to the Flash server on the victim machine is virtually blocked by the ICMP flood. However, the attack has almost no effect on a target system with annotated Linux code, except for the high loss rate for ICMP-echo messages.

[2]Including TCP in rate-limiting does not work because TCP will automatically back-off while other services are trying their hardest to grab bandwidth.

6.3 NIS Server

This section studies ypserv—the yellow page server available on most UNIX systems. Even though the server program itself is simple, it is interesting because it illustrates how different software architectures affect robustness. ypserv is built on top of the RPC protocol [16]. Most RPC programs are built with RPC library and tools like rpcgen, which handles complex tasks such as packaging a call into a message, sending it over the network, and server side message decoding. With the RPC library, the programmer only needs to provide a function that is called when a request arrives. The RPC package is valuable for constructing distributed systems, but it also comes with a potential disadvantage: its virtualization gives programmers less control on the execution of the program.

Linux ypserv-2.2 is a typical RPC server built using these tools. It starts by calling C lib functions svcudp_create, svctcp_create, svc_register and svc_run, which create transport channels, register YP services, and start a server loop that waits for requests. The main service routine ypprog_2 is passed to svc_register as the callback function. ypprog_2 dispatches incoming calls to second level routines such as ypproc_match_2_svc and ypproc_all_2_svc, and sends results back by calling C lib function svc_sendreply.

6.3.1 Claim-and-Hold Attacks

A client program like ypcat requests the entire content of a database from the server. The server handles the request by calling ypproc_all_2_svc. When shipping bulk data over the network, ypserv uses TCP as the transport protocol. We found the same vulnerability to Slow TCP attacks also exists in ypserv. To verify this, we built a customized version of ypcat that uses Slow TCP as its transport. We set up a different number of ypcat attackers, each requesting a database of 150K bytes. While the attack is in progress, we test the server's availability by issuing "rpcinfo -[tu] server ypserv" and normal ypcat commands from a different machine. In addition to the latest version ypserv-2.2, we also tested an earlier version (ypserv-1.3). The main difference between the two versions is that ypserv-1.3 executes ypproc_all_2_svc in a forked child process, and keeps the number of children process below 40. The results are summarized in Table 3, where "Yes" means the normal client successfully gets a response from the server and "No" means the server is unable to reply.

The results show that ypserv-2.2 becomes unresponsive under the presence of *any* slow ypcat attackers. This is not surprising since it is an iterative server that handles only one call at a time. Interestingly, version 1.3 with concurrency support also failed with just 1 slow sender, and damage was done to not only

	ypserv-2.2		ypserv-1.3	
	rpcinfo	ypcat	rpcinfo	ypcat
1 slow sender	No	No	No	No
1 slow reader	No	No	Yes	Yes
40 slow readers	No	No	Yes	No

Table 3: Server Availability under Slow ypcat attacks

TCP but UDP services as well. The reason is that svc_run essentially implements a *poll* loop as in Flash, but using synchronous I/O. When data arrives on a registered channel, the RPC library tries to decode the request message. If the request message is sent slowly, the main server process blocks on a read system call until the entire message arrives. During this time, the server is unable to reply to new requests. The concurrency, however, does help the server survive slow reader attacks, as they are handled by children processes. When the number of slow readers reaches the limit, ypcat starts to fail, but the main process continues to respond to rpcinfo and other YP clients such as ypmatch.

We found that merely annotating ypserv does not give us resilience to Slow TCP attack because the activities we would like to monitor actually occur inside the RPC library rather than the application. Therefore, we really need to annotate the RPC library. However, the effectiveness of doing so is hampered by the library's use of synchronous I/O. We suggest that a more robust RPC library implementation should employ the architecture of the Flash web server, in which (1) low-level stub functions are processed in non-blocking handlers, and (2) user applications like ypserv are invoked as helper processes. If these changes were made, our annotation toolkit would effectively protect the RPC library.

6.3.2 Busy Attacks

There is an easy way to busy attack a ypserv-2.2 NIS server when there is a big database: simply invoke many "ypcat <big database>" simultaneously to ask the service to send the whole database over network. For a database of size 1.7MB, it takes about 20 ms for server to complete the transmission, during which the server does not process any other requests because of RPC's mutual exclusion property. Attacking a NIS server with ypcat flood virtually blocks all NIS operations using TCP, e.g. rpcinfo. Operations that use UDP still go through because they are in a different queue than TCP in select().

We annotated the NIS server by wrapping each NIS operation as a service so that YP_ALL requests (sent by ypcat) will not consume all the resources. An annotated NIS server continues to respond to other YP requests under a ypcat attack, except access to YP_ALL is very slow. However, this is not satisfactory because YP_ALL access to database *group* is required for each log-in. Since *group* is usually a very small database, it is not vulnerable to a ypcat attack. Generally, we do not want to let ypcat attacks on large databases affect access to small databases. Since there are usually only a small number of databases on a NIS server, we can solve this problem by associating a "dynamic" service for each type of operation on each database, so that "ypcat group" and "ypcat passwd" belong to separate services. To support dynamic service, we need to add one new primitive, DYN_SERVICE_ADMISSION(svc_id, min_rate), which is same as SERVICE_ADMISSION except it takes an extra parameter *svc_id* for service id.

7 Limitations

Our approach has several limitations. First, in many cases our approach limits only the scope of damage because it cannot distinguish between "good" and "bad" requests that happen to follow the same code path. In other words, annotations simply augment the classification mechanisms already embedded in the code; they do not add any new ones of their own. To further differentiate between "good" requests and "bad" requests, additional classification mechanisms must be added to the program so that these requests effectively follow different code paths. For cases where separating services according code paths is not fine-grain enough, as we saw in the experiment on the NIS server, we believe that adding DYN_SERVICE_ADMISSION to the toolkit will be necessary. We are currently extending the toolkit to support such a facility.

Second, the current toolkit is only applicable within a single process because the sensors and actuators need to share state, and thus, they work only within a single memory space. This means our toolkit will not work with the current implementation of Apache, for example. It is not clear that an IPC facility can help extend the mechanism to multi-process programs because IPC overhead will likely hinder fine-grain protection. However, for multi-process programs, it is also possible to apply the protection separately for each process. We need more experience to say how effective that will be.

Third, rate-limiting controls only the quantity of resources consumed by each service, but not the order that resources are consumed. Sometimes it is desirable to change the order that we allocate resources, especially when some resource consumers are latency-sensitive. For example, in addition to specifying a rate for all non-TCP packets, we may want to bump TCP packets to the front of the transmission queue. Not being able to schedule resource sometimes forces the user to be more conservative in specifying resource limits. To be able to schedule resource allocation would require support for concurrency within a process, so that the program execution can save the state of the current service task and switch to another service.

8 Conclusions

This paper presents defensive programming as a new approach to offer proactive DoS attack protection. After first identifying two basic types of DoS attacks—busy and claim-and-hold—we build a toolkit that provides an interface programmers use to annotate their code. With compiler assistance, annotations are translated into runtime sensors and actuators that watch for resource abuse and take the appropriate action should abuse be detected. The main strengths of this approach are that it offers fine-grained intra-process protection, can be systematically applied to existing code, protects software from unknown attacks, and puts a minimal burden on the programmer.

Like any mechanism, however, the effectiveness of our approach depends on whether a good defensive policy can be specified, which is the responsibility of the programmer. Our experience with DoS attacks and applications has greatly influenced the design of the annotation interface in order to accommodate the most common policies, but the interface is by no means complete. Also, even with the help of our toolkit, non-trivial programming effort is still required: (1) programmers need to mark service entry points and identify where their programs acquire/release/consume resources, and (2) system administrators need to set system-dependent parameters (e.g., rate limits). Our view is that just as programmers are responsible for making their programs correct, they should also be responsible for making them defensive; we merely provide a set of tools to help simplify this task. Preliminary experience suggests that the programming burden is modest, but we expect to extend and refine the tools as we gain more experience.

Acknowledgments

We would like to thank the anonymous reviewers and Greg Ganger, our shepherd, for helping us improve the clarity and focus of the paper. This work was supported in part by NSF grant ANI-9906704, DARPA contract F30602–00–2–0561, and Intel Corporation.

9 REFERENCES

[1] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System Support for Bandwidth Management and Content Adaptation in Internet Applications. In *Proceedings of the Fourth USENIX Symposium on Operating System Design and Implementation (OSDI)*, Februray 2000.

[2] Apache Software Foundation. Apache Web Server. http://www.apache.org/.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation (OSDI)*, Februray 1999.

[4] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the Fourth USENIX Symposium on Operating System Design and Implementation (OSDI)*, October 2000.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[6] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, May 1996.

[7] K. Kendall. A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems. *Master Thesis, MIT*, June 1999.

[8] Linux NIS(YP) Server. http://www.linux-nis.org/nis/.

[9] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 153–167, October 1996.

[10] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX '99 Annual Technical Conference*, June 1999.

[11] S. E. Perl and W. E. Weihl. Performance assertion checking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 134–145, December 1993.

[12] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[13] C. L. Schuba, I. V. Krsul, M. G. Kuhn, E. H. Spafford, A. Sundaram, and D. Zamboni. Analysis of a Denial of Service Attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Computer Security and Privacy*, May 1997.

[14] A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[15] O. Spatscheck and L. L. Peterson. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation (OSDI)*, Februray 1999.

[16] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. *Request for Comments (RFC) 1831*, August 1995.

[17] C. Villamizar, R. Chandra, and R. Govindan. BGP Route Flap Damping. *Request for Comments (RFC) 2439*, November 1998.

[18] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 2001 IEEE Symposium on Computer Security and Privacy*, May 2001.

[19] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

Using Model Checking to Debug Device Firmware

Sanjeev Kumar*
Department of Computer Science
Princeton University
skumar@cs.princeton.edu

Kai Li
Department of Computer Science
Princeton University
li@cs.princeton.edu

Abstract

Device firmware is a piece of concurrent software that achieves high performance at the cost of software complexity. They contain subtle race conditions that make them difficult to debug using traditional debugging techniques. The problem is further compounded by the lack of debugging support on the devices. This is a serious problem because the device firmware is trusted by the operating system.

Model checkers are designed to systematically verify properties of concurrent systems. Therefore, model checking is a promising approach to debugging device firmware. However, model checking involves an exponential search. Consequently, the models have to be small to allow effective model checking.

This paper describes the abstraction techniques used by the ESP compiler to extract abstract models from device firmware written in ESP. The abstract models are small because they discard some of the details in the firmware that is irrelevant to the particular property being verified. The programmer is required to specify the abstractions to be performed. The ESP compiler uses the abstraction specification to extract models conservatively. Therefore, every bug in the original program will be present in the extracted model.

This paper also presents our experience with using Spin model checker to develop and debug VMMC firmware for the Myrinet network interfaces. An earlier version of the ESP compiler yielded models that were too large to check for system-wide properties like absence of deadlocks. The new version of the compiler generated abstract models that were used to identify several subtle bugs in the firmware. So far, we have not encountered any bugs that were not caught by Spin.

1 Introduction

Device firmware has to be reliable because it is trusted by the operating system. It has the ability to write directly into the physical memory. A stray memory write resulting from a bug can corrupt critical data structures in the operating system and crash the entire machine.

*Now at Microprocessor Research Labs, Intel Corp.

Writing reliable firmware for devices is a challenging problem for three reasons. First, the firmware is implemented using concurrency [25]. And concurrent programs are inherently hard to write correctly. Often, they have unforeseen interactions between the different sequential flows of control resulting in race conditions. Second, event-driven state machines are used to express the concurrency because of their low performance overhead. However, programming with event-driven state machines in languages like C is difficult because they are not designed to support event-driven state-machines programming. Event-driven state-machines programs can be written in these languages using an explicit interface [25] which requires state machines to be specified explicitly using function pointers. The resulting programs are difficult for the programmer to understand and for the compiler to compile efficiently. To get good performance, the programmer is forced to perform some optimizations manually. This introduces subtle bugs in the program. Third, very limited debugging support is available on the devices. Often it is limited to a few memory locations to which the device can write. The programmer has to diagnose the bugs by observing these memory locations on the host machine.

The earlier version of the Virtual Memory-Mapped Communication (VMMC) firmware [14] for Myrinet network interface cards[1] was implemented using event-driven state machines in C. Since the VMMC architecture [5] delivers high-performance on gigabit networks by migrating as much functionality as possible from the operating system to the network interface card, the network interface firmware is fairly complex. Our experience with implementing the VMMC firmware in C has been that while good performance could be achieved, the source code was difficult to write, maintain, and debug. Even after several man-years of debugging, bugs due to race conditions remain and occasionally cause system crashes.

Model checking is a promising approach to building reliable firmware. Model checkers take a model of the system and explore all possible interleaved executions of the concur-

[1]The network interface card has a programmable 33-MHz LANai4.1 processor, 1-Mbyte SRAM memory.

rent system. Since the number of possible executions grows exponentially with the size of the model, abstract models that hide details in the original system are necessary. In addition, often only a fraction of the model can be explored. In spite of these limitations, the systematic search performed by the model checker results in much more extensive testing than traditional methods.

ESP [25] is a language for writing firmware for programmable devices. It uses a model checker to aid in developing and debugging the programs. The language is designed so that its compiler can extract models that can be used by a model checker like Spin [21] to debug the firmware. In the software community, model checking has traditionally been used to find hard-to-find bugs in working systems [7, 15, 27, 30, 18, 17, 22, 11]. In contrast, Spin is used throughout the firmware development process. Usually, the program is developed and debugged entirely using Spin before it is ever run on the device. This is because developing firmware on the device is a slow and painstaking process

Since the version of the VMMC firmware implemented in C was buggy, the firmware was reimplemented using ESP [25]. The ESP compiler extracted models that were very useful in implementing the firmware. A model was used to develop and debug a retransmission protocol in the firmware. It was also used to verify memory safety in the firmware. However, in both of these cases, the models were small because the properties being verified were local and involved only a few ESP processes. The model extracted by this version of the ESP compiler was too big to check for system-wide properties like the absence of deadlocks in the firmware. Since the system-wide bugs are especially difficult for the programmer to find precisely because they are nonlocal, this was a significant limitation of that compiler.

This paper presents techniques used by the new version of the ESP compiler that extracts abstract models. Instead of generating a single model, the compiler now extracts several different models depending on the property that is being checked. These abstract models are significantly smaller because they omit (i.e. abstract away) certain details in the ESP program that are not relevant to the property being verified. This paper also presents our experience with using abstract model to find deadlock bugs in the VMMC firmware. Our main conclusions are as follows:

- The compiler can be used to extract conservative abstract models. In ESP, the abstractions are specified by the programmer. The compiler uses these abstractions conservatively to generate models. Therefore, even if a programmer makes a mistake in specifying the abstraction, every bug in the program will be present in the model. The novelty of this approach is that it gives the programmer control over the abstraction process without relying on the programmer to be correct.

- Abstraction was necessary to generate models that could be used to check for system-wide properties in the VMMC firmware. Using the abstract models, the model checker uncovered seven bugs that would cause the firmware to deadlock. These were subtle bugs that were not caught even after careful code inspection and months of testing and debugging.

- Partial explorations by model checkers can be very effective for debugging. Even using the abstract models, Spin could not exhaustively check the VMMC firmware for deadlocks because of resource constraints. However, in ESP, the model checker is meant to be used as a debugging tool and not to certify correctness. A partial exploration by the model checker uncovered the seven bugs mentioned in the previous paragraph. In addition, we have not encountered any bugs (that were not caught by Spin) while running the firmware on the device.

The rest of the paper is organized as follows. Section 2 presents a brief description of model checkers. Section 3 presents our approach. Section 4 discusses the techniques used by the compiler to generate tractable models. Section 5 describes our experience with using the Spin model checker to develop and debug VMMC firmware. Section 6 discusses related work. Finally, Section 7 presents our conclusions.

2 Model Checking

Model checking is a technique for verifying a system composed of concurrent finite-state machines. Given a concurrent finite-state system, a model checker explores all possible interleaved executions of the state machines and checks if the property being verified holds. A *global state* in the system is a snapshot of the entire system at a particular point in execution. The *state space* of the system is the set of all the global states reachable from the initial global state. Since the state space of such systems is finite, the model checkers can, in principle, exhaustively explore the entire state space.

Model checkers can check for a variety of properties. These properties are traditionally divided into *safety* and *liveness* properties. Safety properties are properties that have to be satisfied in specific global states of the system. Assertion checking and deadlock are safety properties. Assertions are predicates that have to hold at a specified point in one of the state machines. This corresponds to the set of global states where that state machine is at the specified point and the predicate holds. A deadlock situation corresponds to the set of all the global states that do not have a valid next state. Liveness properties are ones that refer to sequence of states. Absence of livelocks is a liveness property because it corresponds to a sequence of global states where no useful work gets done. Liveness properties are specified using temporal logic.

The advantage of using model checking is that it is automatic. Given a specification for the system and the property to be verified, model checkers automatically explore the state space. If a violation of the property is discovered, it can produce an execution sequence that causes the violation and thereby helps in finding the bug.

There are two problems with using model checkers. First, the state space to be explored is exponential in the number of processes and the amount of memory used. So the resources required (CPU as well as memory resources) by the model checker to explore the entire state space can quickly grow beyond the capacity of modern machines. Second, the specification language supported by the model checkers provides limited functionality. So, it is not straightforward to translate concurrent programs written in traditional programming languages into the specification language of the model checkers.

Abstraction is the key to addressing both these problems. Depending on the property be verified, a model that captures only the details relevant to that property needs to be extracted. For properties involving small subsystems, detailed models can be used. However, for properties involving large subsystems, abstract models have to be used.

Models are usually extracted by hand. This process can be time consuming. In addition, it is hard to be sure that the model accurately captures the actual system. Worse yet, as the system evolves, the model has to be independently updated to reflect the changes. Therefore, the use of model checkers is greatly simplified when the models can be extracted automatically[22, 11].

3 Our Approach

Event-driven State-machine Programming (ESP) [25] is a language for programming devices. ESP adopts several structures from the CSP [19] language and has C-style syntax. The basic components of the language are processes and channels. Each process represents a sequential flow of control in a concurrent program and communicates with other processes using rendezvous channels.

ESP is designed to meet three goals. First, ESP should provide language support that makes it easier to develop device firmware. Second, it should allow the use of model checkers like Spin [21] to extensively test and debug the firmware. Third, the compiler should be able to generate efficient executables to run a single processor.

In traditional languages like C, event-driven state-machines programs can achieve high performance by giving up ease of development and reliability. Therefore, they meet only one of the three goals.

To meet all three design goals, the ESP language is designed so that it can not only be used to generate an executable but also be translated into models that can be used by the Spin model checker (figure 1). The ESP compiler takes an ESP program (`pgm.ESP`) and generates 2 types of files. The generated C file (`pgm.C`) can then be compiled together with the C code provided by the programmer (`help.C`) to generate the executable. The programmer-supplied C code implements simple system-specific functionality like accessing device registers to check for network message arrivals. The Spin files (`pgm[1-N].SPIN`) generated by the ESP compiler can be used together with programmer-supplied Spin code (`test[1-N].SPIN`) to verify different properties of the system. The programmer-supplied Spin code generates external events such as network message arrival as well as specifies the properties to be verified.

For each property to be verified, the programmer has to provide test code written in Spin (test[1-N].SPIN in Figure 1). This code is usually fairly small (around 100 lines). Once the test code is written, it can be used to check the ESP program for bugs every time the program is modified.

The earlier version of the ESP compiler [25] generated a single Spin model that included all the details in the ESP program. However, while these models were very useful for checking properties of subsystems consisting of 1-2 processes, they could not be used to check for system-wide properties such as absence of deadlocks. This was due to state-space explosion. Since the hard-to-find bugs are often due to race conditions involving several different processes, ESP now supports automatic extraction of abstract models. Using an abstract model (Section 4), we were able to check for system-wide deadlocks. We found several bugs that resulted in deadlocks (section 5.1).

The ESP approach differs from the previous efforts as follows:

Domain-specific Language ESP is designed not only to simplify the task of programming devices but also to make it easier to extract models. Consequently, any detail in the ESP program that is necessary to check a particular property can be retained in the extracted model. In contrast, general-purpose languages like C, C++, and Java have language features (complex pointer manipulation, exceptions etc.) that are difficult to translate into the specification language of the model checkers [18, 22, 11, 26].

Support for Abstraction Other domain-specific languages [8, 3, 4] extract a single model from the program and use it for model checking. To avoid the state-space explosion associated with detailed models, these languages have been designed to encode only the control structure of the program—the data manipulation is implemented externally in C. In contrast, the ESP language provides support for both control structure and data manipulation. The ESP compiler uses new abstraction techniques to discard some unnecessary details and generate more tractable models.

This paper presents the abstraction techniques used by the new version of the ESP compiler. These techniques are general enough to be applicable to general-purpose languages like C and Java. However, the design of the ESP language makes them particularly effective on ESP programs. For instance, each object in an ESP program can be pointed to by only one of the processes in the program. This limits the amount of pointer aliasing that can occur. Consequently, the increase in state space during abstraction due to aliasing is small (Section 4.2).

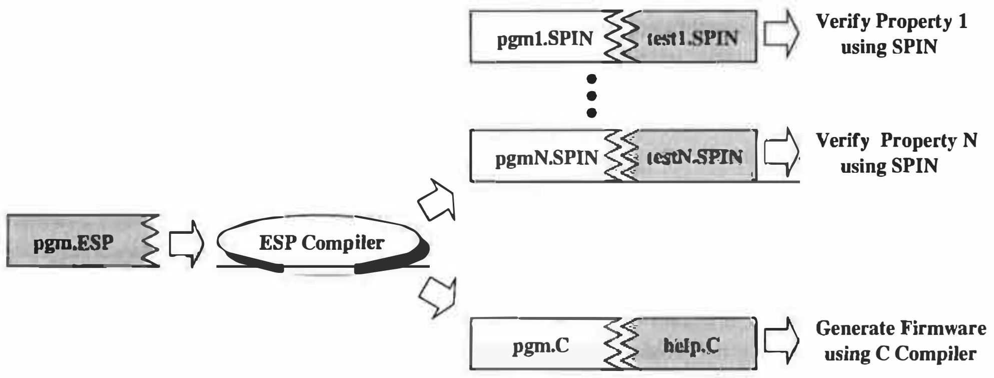

pgm1.SPIN
test1.SPIN
Verify Property 1 using SPIN
pgmN.SPIN
testN.SPIN
Verify Property N using SPIN
pgm.ESP
ESP Compiler
pgm.C
help.C
Generate Firmware using C Compiler

Figure 1: The ESP approach. The shaded regions represent code that has to be provided by the programmer.

3.1 Spin Model Checker

Currently, ESP used the Spin model checker [21]. Spin is a flexible and powerful model checker designed for software systems. Spin supports high-level features like processes, rendezvous channels, arrays, and records. Most other model checkers target hardware systems. Although ESP can be translated into these languages, additional state would have to be introduced to implement features like the rendezvous channels using primitives provided in the specification language. This would make the state explosion problem worse. In addition, the semantic information lost during translation would make it harder for the model checker to optimize the state-space search. Spin allows verification of safety as well as liveness properties.

Spin is a on-the-fly model checker and does not build the global state machine before it can start checking for the property to be verified. So, in cases where the state space is too big to be explored completely, it can do partial searches. It provides 3 different modes for state-space exploration. The entire state space is explored in the *exhaustive* mode. For larger systems, the *bit-state hashing* mode performs a partial search using significantly less memory. It exploits the fact that state spaces are usually sparse and uses a hash function to obtain a much more compact representation for a state. However, since the hash function can map two states onto the same hash, a part of the state space may not be explored. This technique often allows very high coverage (> 98 %) while using an order of magnitude less memory. The *simulation* mode explores single execution sequence in the state space. A random choice is made between the possible next states at each stage. Since it does not keep track of the states already visited, it requires very little memory. However, it could explore some states multiple times while never exploring some other states.

4 Extracting Models Using a Compiler

The ESP compiler generates three types of models: *detailed*, *memory-safety*, and *abstract*. The detailed models contain all the details from the original ESP program. These detailed models often have too much state to be able to perform effective state-space exploration. However, these models are useful during the development and debugging of the firmware using the simulation mode in Spin. They can also be used to check for properties in small subsystems.

The memory-safety models generated are used to check for memory allocation bugs in the program. These models are essentially detailed models with some additional Spin code inserted to check for validity of memory accesses. They contain even more state than the detailed models. In spite of this, these models can be usually used to exhaustively explore the state space for allocation bugs. This is because the memory safety of each individual process can be checked separately using the model checker.

The abstract models generated by the ESP compiler omit some of the details that are irrelevant to the particular property being verified. These models can have significantly smaller state than the detailed models. These models can be used to check for system-wide properties like absence of deadlocks. This class of bugs usually involves several different processes. These bugs are especially hard to find.

Earlier papers [25, 23] described how detailed and memory-safety models are extracted from ESP programs by the ESP compiler.[2] Extracting abstract models requires additional techniques that allow it to abstract away some irrelevant details in the ESP program. These techniques are described in the rest of this section. The simple ESP program in Figure 2 will be used to illustrate the model-extraction process.

In ESP, the programmer specifies the abstractions. The advantage of this approach is that it gives the programmer control over the abstraction process. It allows the programmers to use their understanding about the program and the property being verified to choose the appropriate abstraction. This can result in a better abstraction than ones that can be obtained through static analysis by a compiler.

The ESP compiler uses the abstractions specified by the programmer conservatively when generating the abstract models. Consequently, a bug in the ESP program will always

[2] Briefly, ESP processes and channels are translated into Spin processes and synchronous channels respectively. Since Spin does not support pointers and dynamic allocation while ESP supports them, ESP objects cannot be directly be translated into Spin objects. However, the details of the translation are not necessary for following the rest of this paper.

be present in the abstract model. Our approach is based on a well-known technique[3] of using nondeterminism to broaden the scope of model checking [21, 11, 10]. However, earlier efforts have focused on abstracting simple types like integers. This paper demonstrates how these techniques can be extended to handle complex data types like records, unions, and arrays. This requires addressing additional problems that arise due to pointer aliasing.

4.1 Specifying Abstractions

The abstractions to be performed by the compiler have to be specified by the programmer. The ESP compiler currently allows the programmer to specify two types of abstractions.

Replacing Types. It allows a complex type to be replaced by a much simpler type. This can be done either by specifying an alternative type for the variables individually or by specifying an alternative type in a type declaration. For instance, if the original program contained the following type declaration:

```
type replyT = record of {
  caller: int,
  last: bool,
  addr: int,
  size: int
}
```

then the programmer can specify the following abstraction:

```
replace type replyT = record of {
  caller: int,
  last: bool
}
```

Currently, ESP requires the replacement type to be a supertype of the original type. Essentially, it allows fields from records and unions to be dropped.

Replacing a complex type by a simpler type can significantly reduce the amount of state in the model. For instance, the code to implement the retransmission protocol accepts packets that are implemented as a union of the different types of packets that have to be sent. However, as the content of the packet makes little difference to the correctness of the retransmission code, the complex datatype representing packets can often be replaced by a simple type in the abstract model. Another simplification that can reduce the amount of state is to use smaller arrays than used in the original program. Often, the size of arrays affects only the performance and not the correctness of the program.

Dropping Variables. Some variables that do not affect the validity of a property being checked can be dropped altogether. For instance, a table that keeps track of the mapping between virtual and physical addresses in the main memory might not be relevant when checking the firmware for deadlocks. The variable `table` in process `pageTable` can be dropped by specifying the following abstraction:

```
drop pageTable $table
```

4.2 Extracting Abstract Models

The ESP compiler uses programmer-specified abstractions to generate abstract models. First the compiler performs a type-checking phase during which the compiler determines a type for every expression in the original program (without taking the abstractions into account). The model generator phase can apply the abstractions to each of the statements of this fully-typed program independently.

The abstractions specified can cause some of the expressions in a statement to have an indeterminable value. In these situations, the ESP compiler uses nondeterminism to make conservative approximations that strictly generalizes the scope of model checking. The various expressions in a statement can be classified into two classes: *left-exp* and *right-exp*. They are handled as follows:

left-exp. A left-exp is an expression that is used to determine a memory location to which a value will be stored. These expressions appear on the left side of the assignment statements and in `in` operations on channels. Consider the following statements:

```
a = b;
a[i].last = d;
```

where variable a has the type

```
type tableT = #array of #record of {
  first: int, last: int
}
```

In the simplest cases, when a left-exp becomes undeterminable, the statement can be simply discarded during model extraction. For instance, if the variable a is dropped by the abstraction, the first statement `a = b;` becomes irrelevant and can be discarded. This is because the only side effect of that statement is to the variable a. Similarly, if the `last` field is dropped from the type `tableT`, the second statement can be discarded during model extraction. This is because all objects of that type no longer have the `last` field. As a result, the statement has no remaining side effect in the generated model.

The most general case that has to be handled occurs in the second statement when variable a or variable i is dropped.

[3]When performing abstractions, some of the values in the model might become *undeterminable*. For instance, the value of a variable in the model that depended on another variable in the original program that was discarded during abstraction will become undeterminable. The compiler keeps track of these values and makes sure that the abstract model only broadens the scope of model checking. For example, when the value of a condition in a conditional statement cannot be determined, it can be replaced by a nondeterministic choice in the model. During model checking, both the branches of the conditional statement will be explored. Broadening the scope of model checking can introduce spurious bugs (false positives) in the abstract model. However, all the bugs that were present in the program will be retained in the model.

```
#define TABLE_SIZE     100
#define PAGE_SIZE      4096
#define PAGE(a)        ( (a) / PAGE_SIZE)
#define OFFSET(a)      ( (a) % PAGE_SIZE)
#define ADDR(p)        ( (p) * PAGE_SIZE)

type reqT = record of { caller: int, addr: int, size: int}
type replyT =  record of { caller: int, last: bool, addr: int, size: int}
channel reqC : reqT
channel replyC : replyT

process pageTable {
  $table : #array of int = #{ TABLE_SIZE ~> 0 ...};
  // Omitted : Code to initialize the table
  while ( true) {
    in( reqC, { $caller, $vaddr, $size});    // Get the next request
    assert( !OFFSET(vaddr));      // Assumes vaddr is page aligned
    $done: bool = false;
    while( !done) {
      $paddr : int = ADDR(table[PAGE(vaddr)]); // Lookup physical address
      $chunk : int = PAGE_SIZE;     // Calculate the size
      if ( size < PAGE_SIZE)    chunk = size;
      size = size - chunk;
      done = ( size == 0);
      out( replyC, { caller, done, paddr, size}); // Send a reply
    }
  }
}

process transfer {
  while ( true) {
    // Omitted : Code that generates values for variables 'vaddr' and 'size'
    out( reqC, { @transfer, vaddr, size});
    $last : bool = false;
    while( !last) {
      in( replyC, { @transfer, last, $paddr, $chunk});
      // Omitted : Code to transfer 'chunk' bytes at 'paddr'
    }
  }
}
```

NOTES

1. Process `pageTable` translates virtual addresses into physical addresses. It maintains a table that maps virtual page numbers into physical page numbers. It accepts translation requests on channel `reqC` and sends replies on channel `replyC`. Since a region of contiguous virtual memory can map onto a set of noncontiguous physical pages, each request sent on channel `reqC` can yield multiple replies on channel `replyC`. The last reply is identified by the `last` field in the reply.
2. Process `transfer` computes a pair of `vaddr` and `size` that identifies a region in virtual memory. It sends a request on `reqC` to translate it into physical addresses. It then receives the physical addresses on channel `replyC` and uses it to transfer data.
3. Since other processes might be sending requests on channel `reqC`, the caller field on the channels `reqC` and `replyC` is used to match the replies with the request. `@transfer` is a constant that represents the process id for the process `transfer`.

Figure 2: A ESP program.

In this case, the object pointed to by `a[i]` is being mutated, and the change would be visible to any other pointer that was pointing to the same object. To handle this case, the compiler has to determine a list of pointers to which `a[i]` could be aliased. For each of these pointers, the generated model has to include a nondeterministic statement that either updates the object to which it points or does not update that object.

Nondeterministically updating a large set of objects can dramatically increase the amount of state space that has to be explored. It can also result in false-positive bugs being introduced into the model. A number of techniques can be used to narrow the list of pointers to which `a[i]` can be aliased. First, only pointers of the same type as `a[i]` have to be considered. Second, only pointers within the same process can be aliased to `a[i]`, since processes in ESP do not share objects. Third, in the case where only variable `i` is dropped, only objects pointed to by an entry in array `a` needs to be considered. Finally, alias analysis can be used to further reduce the list of pointers. If compile-time analysis can determine that the pointer `a[i]` is not aliased to any other pointer, the situation reduces to the simple case where the statement is simply discarded.

Pattern matching also needs to be performed conservatively. We will illustrate this using code shown in Figure 3. In ESP, a pattern can appear on the left-hand side of an assignment statement or in an `in` statement. For instance, the `in` operation on channel `patternC` in process `A` uses a pattern. The pattern specifies that it expects the value 5 in the `caller` field of the record it receives on the channel. Consequently, the pair of `in` and `out` operations will succeed only if the `out` operation in process `B` supplies the value 5 in the `caller` field.

Suppose the programmer uses the abstraction in Figure 3 on the ESP program in the same figure. The abstraction drops the `caller` field on the channel. In the absence of the `caller` field in the extracted model, there is no way to determine if the pattern matching on the channel should succeed. Therefore, a nondeterministic statement is inserted in process `A` before the `in` operation on the channel. The `in` operation will succeed only if the variable `match` is set to `OK`. This captures both cases: the case in which the `in` operation on channel `patternC` would have succeeded and the case in which it would not have succeeded.

Being conservative on patterns ensures that a deadlock state is represented in the extracted model. A deadlock state could get translated into a live state if an exit transition is added to that state during model extraction. ESP avoids this by ensuring that each state in the program in which a process could be blocked on a particular channel is represented by a state in the extracted model where that process is blocked on that channel.

right-exp. All expressions that are not left-exp expressions are right-exp expressions. These generate values to be used at various points in the programs. They appear on the right side of the assignment statements, in conditionals of `if` and `while` statements, and in `out` operations on channels.

ESP Program

```
type patternT = record of {
  int caller, int count
}
channel patternC : patternT
process A {
  in( patternC, { 5, $count});
}
process B {
  // Omitted: code to declare and initialize 'caller'
  out( patternC, { caller, 45});
}
```

Abstraction

```
replace type patternT = record of {
  int count
}
```

Extracted Spin Model

```
mtype = { OK, DONT_SEND, DONT_RECV};
typedef patternT { int nil; int count; };
chan patternC = [0] of { mtype, patternT };
proctype A() {
  mtype match; int count;
  // Nondeterministically pick a value for 'match'
  if
  :: skip -> match = OK
  :: skip -> match = DONT_RECV
  fi;
  patternC ? eval(match), 0, count
}
proctype B() {
  patternC ! OK, 0, 45
}
```

Figure 3: A program to illustrate the handling of patterns. For simplicity, some details that are unnecessary to understand this example have been omitted.

During abstraction, the value of a right-exp expression can become undeterminable. Ideally, these expressions should be replaced by one that nondeterministically returns a valid value of the type of the expression. This will cause the model checker to try all possible valid values during the state space exploration.

For boolean expressions, only two values are possible and a nondeterministic choice between the two can be made. So, a boolean expression in a conditional statement (like an `if` statement) whose value can no longer be computed is replaced by a nondeterministic statement [21]. During model checking, both the branches of the conditional statement will be explored.

For nonboolean expressions, trying all possible valid values would be computationally very expensive during model checking. It is also usually unnecessary because a small set of values can effectively cover the entire space. However, there is no general way for the ESP compiler to determine the set of values that would be sufficient to cover the entire state space. Therefore, the ESP compiler relies on the programmer to supply the right set of values. For each variable in the program (except boolean variables) for which the abstract model needs a nondeterministic value, a channel is generated in the abstract model. When a value is needed, the model performs a read operation on the channel. The programmer is responsible for supplying values on the channel using a nondeterministic statement. For instance, the following code executes an infinite do loop and nondeterministically supplies either of the three values (5, 6, and 9).

```
do
:: intC ! 5
:: intC ! 6
:: intC ! 9
od
```

For nonscalar types like arrays and records, the set of values that the programmer has to provide on these channels not only includes new objects but also all existing objects in the model to which it could be aliased.

4.3 Discussion and Limitations

Size of State Space. In principle, an abstract model generated can have more states than the corresponding detailed model. This is because two different things happen during abstraction. First, some details in the program are discarded. This will reduce the number of states. Second, the compiler uses the abstractions specified conservatively by translating some of the deterministic statements in the program into nondeterministic statements. This can increase the number of states that have to be explored.

In practice, the abstract model has significantly fewer states. This is because only a small number of nondeterministic statements get introduced. In addition, since the programmer specifies the abstraction in ESP, the programmer can pick the abstraction carefully so as to minimize the number of states. For instance, if the packet sequence number in the implementation of a network retransmission protocol is unnecessary to verify a particular property, the programmer can specify that all variables and fields of records that store sequence numbers in the program should be discarded from the abstract model. Then, the model will no longer have any statement that uses the sequence number. Consequently, no new nondeterministic statements will be introduced into the model due to this abstraction.

Bugs. By being conservative, ESP ensures that all bugs in the original program are present in the extracted model. However, this does not guarantee that all bugs will be caught during model checking. First, the model checker might not be able to check the entire state space because of resource constraints. Second, ESP relies on the programmer to provide test code (Section 3) as well as code that generates values on some channels (Section 4.2). A mistake by the programmer can result in bugs being missed during model checking.

The techniques described in this paper help reduce and isolate the portion of Spin code where mistakes can be made by a programmer. This is analogous to type-safe languages that rely on C to implement unsafe portions of a program. In this case, a program can have type-safety errors. However, these errors are isolated in the portion of the program that is implemented in C.

Abstraction 1

```
replace type reqT = record of {
  caller: int
}
replace type replyT = record of {
  caller: int,
  last: bool
}
drop pageTable $table, $vaddr, $size
drop pageTable $paddr, $chunk
drop transfer $vaddr, $size, $paddr
drop transfer $chunk
```

Abstraction 2

```
replace type reqT = record of {
  caller: int
}
replace type replyT = record of {
  caller: int
}
drop pageTable $table, $vaddr, $size
drop pageTable $paddr, $chunk, $done
drop transfer $vaddr, $size, $paddr
drop transfer $chunk, $last
```

Figure 4: Two Abstractions for the ESP program in Figure 2

4.4 Example

In this section, we will use an example to illustrate the use of abstraction to check for a property. We will check for absence of deadlocks in the ESP program in Figure 2.

'Abstraction 1' in Figure 4 can be used to check the program for the absence of deadlocks. Using the abstraction, the ESP compiler generates an abstract model (Figure 5). The abstraction drops all variables except `caller` and `done` in process `pageTable` and `last` in process `transfer`. It also replaces the types of the two channels. During abstraction, the value of the boolean variable `done` becomes indeterminate because its value depends on the value of the variable `size` that was dropped. The compiler translates the statement

```
    done = ( size == 0);
```

into Spin code that nondeterministically assigns either values `true` or `false` to it as follows:

```
    if
    :: skip -> done = 0
    :: skip -> done = 1
    fi
```

The Spin model checker can exhaustively explore the entire state space (12 states!) and determine that there are no deadlocks. In contrast, if a detailed model was used, the model checker would have to potentially explore a large number of states (by trying all possible values for `vaddr` and `size`) to determine that there were no deadlocks.

Since the compiler makes conservative approximations when generating abstract models, the model checker will not miss a deadlock because of a programmer error in specifying an abstraction. However, a programmer error can cause a spurious deadlock to be flagged. For instance, 'Abstraction 2' in Figure 4 will detect a spurious deadlock because the programmer dropped the variable `done` by mistake. These will have to be double checked by the programmer.

Finally, we can introduce a deadlock in the program by replacing the line

```
    $done: bool = false;
```

by the line

```
    $done: bool = ( size == 0);
```

This will cause a deadlock if the `size` specified on channel `reqC` is 0. The model checker will find the bug using either of the two abstractions.

5 Debugging VMMC Firmware

The VMMC firmware was reimplemented using ESP. The earlier implementation in C includes about 15600 lines of C code. In contrast, the new implementation using ESP required 500 lines of ESP code together with around 3000 lines of C code. The C code implements simple tasks like initialization, initiating DMA, packet marshalling and unmarshalling and shared data structures with code running on the host processor (in the VMMC library and the VMMC driver). All the complex state machine interactions are restricted to the ESP code, which uses 8 processes and 19 channels. This is a significant improvement over the earlier implementation where the complex interactions were spread throughout the 15600 lines of hard-to-read code.

The previous version the compiler was used to extract detailed and memory-safety models that were useful in checking for local properties involving 1-2 processes [25, 23]. First, Spin was used throughout the development process. In the software community, model checking has traditionally been used to find hard-to-find bugs in working systems. However, since developing firmware on the network interface card involves a slow and painstaking process, Spin was often used to develop code. For instance, the retransmission code, which uses a simple sliding window protocol with piggyback acknowledgement, was developed and debugged entirely using the Spin simulator. Once debugged, the firmware was ported to the network interface card with little effort. Second, Spin was used to verify memory safety in the VMMC firmware. Instead of supporting safety through garbage collection, ESP provides an explicit `malloc/free`-style interface to support dynamic memory management. Although this interface is unsafe, the Spin model checker can be used to verify memory safety. To facilitate this, ESP uses a novel scheme that makes memory safety a local property of each process. This allows each process to be verified separately resulting in smaller models. Consequently, memory safety of the VMMC firmware could be checked exhaustively.

File	Lines of Code
ESP Program	453
Abstraction Specification	108
Test Code	128
Abstract Model Extracted	2202

Table 1: Sizes (in lines) of the various files used to debug the VMMC firmware. The first three files have to be provided by the programmer while the last one is generated by the ESP compiler.

The size of the models generated by the previous version of the ESP compiler was too large to check for system-wide properties like absence of deadlocks. In the rest of this section, we describe our experience with using the abstract models generated by the new version of the compiler to check for deadlocks in the VMMC firmware.

5.1 Deadlocks in VMMC Firmware

System-wide deadlocks are often a result of complex interactions in the program and can be difficult for programmers to find. Therefore, the use of model checking to find these bugs is important. We used an abstract model to find bugs that would cause the firmware to deadlock.

Table 1 shows the sizes of the various files that were used to find bugs with the abstract model. The abstraction specification specifies 63 variables to be dropped (1 line each in the specification), replaces the type of one variable by a simpler type, and replaces 18 types by simpler types. It is fairly easy for the programmer to identify the parts of the program that should be abstracted away. In the VMMC implementation, only a handful of the variables required closer examination to decide whether or not they needed to be abstracted. The entire abstraction specification took a few hours to write.

Using the abstract model, Spin found several subtle bugs in the VMMC firmware that would cause the firmware to dead-

```
mtype = { OK, DONT_SEND, DONT_RECV};
typedef reqT { int nil; int caller; };
typedef replyT { int nil; int caller; bool last; };
chan reqC [NUM_PROCESSES] = [0] of { mtype, reqT };
chan replyC [NUM_PROCESSES] = [0] of { mtype, replyT };

proctype pageTable( int pid) {
  int caller; bool done;
  atomic {
    do
    :: 1 -> {
         reqC[ pid] ? OK, 0, caller;
         done = 0;
         do
         :: ( !done) -> {
              // Nondeterministically assign a value to 'done'
              if
              :: skip -> done = 0
              :: skip -> done = 1
              fi;
              replyC[pid] ! OK, 0, caller, done
            }
         :: else -> break
         od
       }
    od
  }
}
proctype transfer( int pid) {
  bool last;
  atomic {
    do
    :: 1 -> {
         reqC[pid] ! OK, 0, 1;
         last = 0;
         do
         :: ( !last) -> replyC[pid] ? OK, 0, 1, last
         :: else -> break
         od
       }
    od
  }
}
```

Spin NOTES

1. `'?'` and `'!'` are used to receive and send messages on channels. Constants specified in a receive operation has to match the corresponding values in the send operation for the data to be successfully transferred on a channel.
2. `if` and `do` are nondeterministic statements. They differ in that an `if` is executed only once while a `do` is executed repeatedly until a `break` statement is executed in the body. These statements become deterministic (and turn into regular `if` and `do`) statements when there is only one choice or when there are two choices and one of them is an `else` statement.
3. `mtype` is like `enum` in C.
4. `@transfer` in Figure 2 gets translated into the constant `1`.
5. Some details like the `atomic` statement, the `mtype` field, and the `nil` fields can be ignored. They are not needed to understand this example.

Figure 5: Abstract Spin model generated from the ESP program in Figure 2 using 'Abstraction 1' in Figure 4

Spin Search Mode		Exhaustive	Bit-state hashing
Limiting Resource		Memory	CPU Time
No. of States	Stored	117351	22574700
	Matched	265492	77165900
Time (hr:min:sec)		0:01:24	3:57:30
Memory (in MBytes)		268.35	167.92

Table 2: Checking for the absence of deadlocks in the VMMC firmware using Spin. In both Spin modes, the state-space exploration could not be completed because of resource constraints. The *stored* column shows the number of unique states encountered while the *matched* column shows the number of states encountered that had already been visited before.

lock. However, even with the abstract model, an exhaustive search of the state space was not possible because of resource constraints. Therefore, only partial searches were performed. After the bugs that were found were fixed, further state-space exploration using Spin did not uncover any more bugs. Table 2 presents the amount of state space that could be explored using the resources available. In the exhaustive mode, Spin had to abort the search after 84 seconds because it ran out of memory. In the bit-state hashing mode, Spin ran for 3 hours and 57 minutes before the search was terminated by the user.[4] Since Spin only could perform a partial search, we cannot conclude that there are no more bugs in the VMMC firmware. However, we have not encountered any bugs while running the firmware on the device that were not caught by Spin.[5]

Even using a partial search, Spin found seven bugs in the firmware. These were subtle bugs that were not caught even after careful code inspection and months of testing and debugging. The VMMC firmware in ESP was designed to avoid the bugs encountered in the earlier implementation of the firmware in C. In addition, the ESP language allowed the complex interactions in the system to be implemented concisely (around 500 lines). In spite of this, the model checker uncovered several bugs that could deadlock the system. This highlights the limitations of careful code inspection and traditional testing, and the benefits of using tools like model checker that explore the various possible scheduling scenarios systematically.

The first bug was due to a circular dependency involving 3 processes that resulted in a deadlock. Once identified, the deadlock was avoided by eliminating the cycle.

The second bug involved a situation when the sliding window in the retransmission protocol was full and, therefore, not accepting any new messages to be sent to the network. This eventually led to no new data packets being accepted from the network. Since incoming messages were delivered in FIFO order, an explicit acknowledgement message that could unlock the system was trapped behind a data packet resulting in a deadlock. To fix this problem, packets have to be dropped occasionally to allow the explicit acknowledgements to get through.

Two other bugs uncovered were similar to the bug that was discussed in the example in Section 4.4. They would result in deadlocks if applications requested zero-byte data transfers.

The remaining bugs discovered involved receiving unexpected messages or not receiving expected messages. The first bug involved receiving acknowledgments with invalid acknowledgement numbers. This was fixed by first checking for the validity of the acknowledgement numbers before using them. The second bug involved receiving an unexpected import reply message. These messages are usually received in response to an import request. An unexpected reply would deadlock the system. The problem was fixed by adding code that discarded these unexpected messages. The final bug involved not receiving a reply to an import request. We had been aware of this bug but had not fixed it yet. This bug can be eliminated using a timeout.

5.2 Discussion and Limitations

The model checker is very effective in finding bugs. Spin was used to develop and debug the VMMC firmware implementation in ESP before running it on the device. So far, we have not encountered any bugs that were not caught by Spin. This is in contrast with our earlier firmware implementation in C where we encountered new bugs every time we tried a different class of applications or ran on a bigger cluster.

Model checking allows bugs to be uncovered early in the debugging process. This is highlighted by the fact that several bugs found by Spin would not have been discovered using conventional testing as long as our VMMC implementation was used on all the machines in the network. These bugs could only be triggered when the firmware was used to communicate with other VMMC implementations that generated unexpected messages because they were either malicious or buggy.

Partial searches are very effective in finding bugs in the concurrent programs. This is because the state machine being explored is usually much larger than necessary in which each state of the minimal state machine is represented multiple times. Techniques like abstraction and optimizations like partial-order reduction [20] try to eliminate some of this redundancy. However, significant redundancy remains because the size of the state space is exponential in the size of the model. For instance, a variable `i` whose value ranges from one to ten but has no bearing on the property can result in each state of the minimal state machine being explored ten times. So even a partial search that explores a small fraction of the state space can cover a significant fraction of the minimal state machine.

The effectiveness of using abstract models to check larger

[4]The Spin model checker was run overnight on a shared machine for over 12 hours. 3 hours and 57 minutes was the processor time that the model checker was allocated during this period.

[5]The firmware was used to run a number of microbenchmarks and applications on a 16 processor (4x4) cluster [24].

systems is demonstrated by the fact it could be used to find bugs in the VMMC firmware. Unfortunately, it is difficult to quantify the actual reduction in the size of the state space that resulted from abstraction or the fraction of the state space that was explored by the partial search. This is because the actual size of the state space can only be determined by exploring the entire state space. It is possible to compute an upper bound on the size of the state space.[6] However, this grossly overestimates the size of the state space because the state space tends to be very sparse.

One of the limitations of model checking is that it requires test code to be provided for each property to be verified. The test code is responsible for simulating the environment (external events like network message arrivals) as well as specifies the property to be verified. A mistake in the test code can result in the wrong property being checked or, more commonly, failing to explore a part of the state space for bugs. The latter happens when the environment is over constrained. For instance, when debugging the retransmission protocol, our initial code simulated a well-behaved environment that only generated network packets with the expected sequence numbers. Later, we found bugs when the test code was modified to generate unexpected messages. Another problem that we encountered when looking for deadlocks was that a deadlock that involved only a few processes in the model could go undetected. This is because a deadlock in Spin is a state out of which there are no transitions. When a deadlock involves only a few processes, the remaining processes can sometimes make progress. Consequently, there are always transitions out of the current state. Therefore, the state machine is not technically deadlocked. For instance, this happens when a deadlock in one part of the system prevents messages from being sent while another part of the system responsible for receiving messages from the network continues to function correctly. To avoid this, the test code can be changed to ensure that no part can inactive for long periods of time (by maintaining activity counters).

6 Related Work

6.1 Model Extraction Approaches

Model extraction by hand. Several researchers have verified various aspects of Operating Systems using model checkers. These efforts involved extracting an abstract model of the system by hand. Spin was used to verify the Interprocess Communication Subsystem in Harmony [7] (a real time operating system) and RUBIS microkernel [15]. The latter study found that significant effort was needed in extracting the model. Spin was also used to develop and verify a synchronization protocol for Plan 9 [27]. More recently, Spin was used to verify the IPC system of the Fluke OS [30]. All these studies found that the model checker was able to find some hard to find race conditions.

[6]If n is the number of bits needed to encode a state, then 2^n is an upper bound.

Automatic Model Extraction. To avoid the problems with model extraction by hand, some researchers have extracted the models automatically from the source code. Teapot [8] is a domain-specific language for implementing software cache coherence. It extracts a model that can be used by the Murphi model checker [13]. Promela++ [3] is a language for implementing layered protocol. Its compiler generates model that can be used by the Spin model checker. Esterel [4] is a language for specifying synchronous reactive systems and is primarily used in hardware design. The Esterel programming environment includes verification tools like model checkers that can be used to test the programs. Esterel was used to implement a subset of the TCP protocol [6]. They showed that esterel could be used to generate efficient code. However, they did not report any experience with the verification tools.

In all these cases, the domain-specific language is used to encode the control structure of the program. The rest of the program (data handling) is handled using a different language (typically C). The compiler for these languages extracts a single model that reflects the control structure of the program.

Java PathFinder [18] translates Java programs into Spin specifications. It handles significant subset of the Java including dynamic object allocation, object reference, exception processing and inheritance. However, they do not handle features like method overriding and overloading. Also, they do not provide a way to abstract details so that a tractable model can be extracted.

Verisoft [17] uses a different approach to perform model checking on a concurrent system. Instead of trying to extract a model, it explores the state-space of the system by replacing the scheduler of the concurrent system. By controlling the scheduler, it can force the concurrent program to execute all possible interleavings. This allows it to apply model checking to actual programs written in traditional languages like C (instead of a model). The problem is that it can explore much smaller state spaces because some of the techniques used by model checkers like Spin to optimize the exploration cannot be applied.

Automatic model extraction with support for abstraction. More recent efforts have focused on extracting several abstract models to verify different properties in the system.

FeaVer [22] extracts Spin models from programs written in a C dialect that has simple extensions to support event-driven state machines. The system allows the programmer to specify pairs of C and Spin code patterns. When the C pattern is encountered during translation, the corresponding Spin code is generated. This approach automates the extraction of abstract models. However, the translator does not have any semantic information to check the validity of the translation. The system was used to debug the call processing software for Lucent's Pathstar access server.

Lie et. al. [26] use an approach similar to FeaVer [22] to extract Murphi [13] models from C programs. It requires the programmer to specify two things: a set of patterns that identify the C code that has to be captured in the extracted model, and transformations that translate the identified C code into Murphi code. Unlike FeaVer, it uses program slicing [31, 29] to extract additional code that affects the identified code. However, the standard slicing algorithms have problems with C constructs like pointers, unions and unstructured control flow. Like FeaVer, it cannot check the validity of the generated model.

Bandera [11] allows automatic extraction of finite state models from Java programs. It uses techniques like program slicing [31, 29] and data abstraction to allow more tractable models to be extracted. ESP and Bandera differ in how nondeterminism is used during model extraction. In Bandera, nondeterminism is used only when an *undeterminable* value flows into a test of a conditional statement. In ESP, nondeterminism is used to assign values to all variables and fields that are not dropped by the abstraction but whose values become *undeterminable*. Another difference between ESP and Bandera is that the ESP language was designed to permit model checking. In contrast, Bandera targets Java that has a number of language features that are difficult to translate efficiently into models. So far, Bandera has been used to verify properties in only simple programs.

The SLAM project [1, 2] extracts a predicate abstraction to check assertions in sequential programs written in C. A predicate abstraction is a model with only boolean variables that correspond to conditions in the original program. The assertion is checked in the predicate abstraction using a model checker. Since the checker may generate false positives, symbolic execution is used to verify the counter examples generated by the model checker. If the counter example is invalid, the predicate abstraction is refined to eliminate the counter example. This approach has not yet been extended to handle concurrent programs.

6.2 Debugging System Software

A vast amount of research has focused on the problem of debugging system software. The techniques used span language design, model checking, compiler analysis, and runtime methods. In this section, we discuss some of the related work in this area.

As described in Section 6.1, model checkers have been used to debug system software. Some [7, 15, 27, 30, 18, 17, 22, 11] have focused on debugging programs written in general purpose languages like C, C++ and Java. Others have proposed domain-specific languages [8, 3, 4] that have been designed with model checking in mind, and therefore, allow model checking to be more effective.

Meta-level Compilation [9, 16] provides a framework for extending a compiler with application-specific code that can be used to statically check certain properties of that application. It was used to look for bugs in several systems including the cache coherence protocols for the FLASH multiprocessor and the Linux kernel. This technique requires little change to the source code and was very effective in finding several hundred bugs in these systems. The compiler extensions look for violations of properties like proper buffer allocation and deallocation, and absence of deadlocks. However, these extensions perform only intra-procedural analysis. In some instances, a separate global pass was used to combine data gathered by the intra-procedural analysis of the different functions to check a global property. Since the static analysis is inexact, it can generate false-positives. The bugs reported have to be double-checked by the programmer. In addition, the limited scope of intraprocedural analysis can generate false-negatives.

Eraser [28] detects data races in multithreaded programs. It instruments the program binary to check at runtime that a lock protects each shared variable access. It does not impose any constraints on the programs, and therefore works on existing programs with little modifications. However, the tool can detect only the data races that occur during the debugging runs; it is the programmer's responsibility to ensure that the program is run with several different inputs so that it is tested thoroughly. In addition, the instrumentation results in a factor of 10 to 30 slowdown in program execution. This can prevent some data races in the program from occurring during debugging.

Programming language features can often prevent an entire class of bugs. Safe programming languages prevent a program from accessing a dynamically allocated object after it has been freed. The Vault [12] language uses an expressive type system to enforce high-level protocols in system software. The type system allows a module writer to specify properties like "a read system call to read from a file can be called only after that file has been opened using the open system call".

7 Conclusions

ESP allows abstract models to be extracted from the programs. These models are smaller because they omit details that are irrelevant to the property being checked. In ESP, the programmer specifies the abstraction and therefore has control over the abstraction process. The ESP compiler uses the abstractions specified by the programmer conservatively when generating an abstract model. This ensures a bug in the ESP program will be in the generated model even when a programmer makes a mistake in specifying the abstraction.

Abstraction was essential for obtaining models that could be used to check for system-wide properties like absence of deadlocks. The earlier version of the compiler [25] was unable to find deadlock bugs in the VMMC firmware. The new version of the ESP compiler generated an abstract model that was successfully used to uncover seven deadlock bugs. Even with these models, only a partial search was possible. In spite of this, we have not encountered any new bugs while running the firmware on the device.

The use of model checker is greatly simplified when the models could be automatically extracted from the ESP programs by the compiler. Other studies [22, 11] have shown that automatic extraction is possible even for traditional languages like C and Java. This not only increases our confidence that the model accurately reflects the program but also allows the system to be rechecked with little effort whenever changes are made to it.

A model checker is very effective as a debugging tool. Sometimes, only a partial state-space exploration is possible due to resource constraints. However, this is acceptable because the goal is to identify bugs and not to certify correctness. Even a partial systematic search by the model checker results in more extensive testing than traditional testing methods and can be invaluable in debugging concurrent firmware.

Acknowledgments

This work was supported in part by the National Science Foundation (CDA-9624099,EIA-9975011,ANI-9906704,EIA-9975011), the Department of Energy (DE-FC02-99ER25387), California Institute of Technology (PC-159775, PC-228905), Sandia National Lab (AO-5098.A06), Lawrence Livermore Laboratory (B347877), Intel Research Council, and the Intel Technology 2000 equipment grant.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Programming Languages Design and Implementation*, 2001.

[2] T. Ball and S. K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *International Spin Workshop*, 2000.

[3] A. Basu, T. von Eicken, and G. Morrisett. Promela++: A Language for Correct and Efficient Protocol Construction. In *Infocom*, 1998.

[4] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

[5] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *International Symposium on Computer Architecture*, 1994.

[6] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating Efficient Protocol Code from an Abstract Specification. In *SIGCOMM*, 1996.

[7] T. Cattel. Modeling and Verification of a Multiprocessor Realtime OS Kernel. In *International Conference on Formal Description Techniques*, 1994.

[8] S. Chandra, B. E. Richards, and J. R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Programming Languages Design and Implementation*, 1996.

[9] A. Chou, B. Chelf, D. Engler, and M. Heinrich. Using Meta-level Compilation to Check FLASH Protocol Code. In *Architectural Support for Programming Languages and Operating Systems*, 2000.

[10] C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically Closing Open Reactive Programs. In *Programming Languages Design and Implementation*, 1998.

[11] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Shawn, and L. Hongjun. Bandera: Extracting finite-state models from Java source code. In *International Conference on Software Engineering*, 2000.

[12] R. DeLine and M. Fahndrich. Enforcing High-Level Protocols in Low-Level Software. In *Programming Languages Design and Implementation*, 2001.

[13] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[14] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Hot Interconnects*, 1997.

[15] G. Duval and J. Julliand. Modeling and verification of the RUBIS μ-Kernel with Spin. In *International Spin Workshop*, 1995.

[16] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Operating Systems Design and Implementation*, 2000.

[17] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Principles of Programming Languages*, 1997.

[18] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. In *International Journal on Software Tools for Technology Transfer*, 1999.

[19] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[20] G. Holzmann and D. Peled. An Improvement in Formal Verification. In *International Conference on Formal Description Techniques*, 1994.

[21] G. J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[22] G. J. Holzmann and M. H. Smith. A Practical Method for Verifying Event-Driven Software. In *International Conference on Software Engineering*, 1999.

[23] S. Kumar and K. Li. Dynamic Memory Management for Programmable Devices. In *International Symposium of Memory Management*, 2002.

[24] S. Kumar and K. Li. Performance Impact of Using ESP to Implement VMMC Firmware. In *Workshop on Novel Uses of System Area Networks (SAN-1)*, 2002.

[25] S. Kumar, Y. Mandelbaum, X. Yu, and K. Li. ESP: A Language for Programmable Devices. In *Programming Languages Design and Implementation*, 2001.

[26] D. Lie, A. Chou, D. Engler, and D. Dill. A Simple Method for Extracting Models from Protocol Code. In *International Symposium on Computer Architecture*, 2001.

[27] R. Pike, D. Pressoto, K. Thompson, and G. Holzmann. Process sleep and wakeup on shared-memory multiprocessors. In *EurOpen Conference*, 1991.

[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *Transactions on Computer Systems*, 15(4):391–411, 1997.

[29] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[30] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: A practical tool for OS implementors. In *Hot Topics in Operating Systems*, 1997.

[31] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.

CMC: A Pragmatic Approach to Model Checking Real Code

Madanlal Musuvathi,[*] David Y.W. Park,[†] Andy Chou,
Dawson R. Engler, David L. Dill

{madan, parkit, acc, engler, dill}@cs.stanford.edu
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A

Abstract

Many system errors do not emerge unless some intricate sequence of events occurs. In practice, this means that most systems have errors that only trigger after days or weeks of execution. Model checking [4] is an effective way to find such subtle errors. It takes a simplified description of the code and exhaustively tests it on all inputs, using techniques to explore vast state spaces efficiently. Unfortunately, while model checking systems code would be wonderful, it is almost never done in practice: building models is just too hard. It can take significantly more time to write a model than it did to write the code. Furthermore, by checking an abstraction of the code rather than the code itself, it is easy to miss errors.

The paper's first contribution is a new model checker, CMC, which checks C and C++ implementations directly, eliminating the need for a separate abstract description of the system behavior. This has two major advantages: it reduces the effort to use model checking, and it reduces missed errors as well as time-wasting false error reports resulting from inconsistencies between the abstract description and the actual implementation. In addition, changes in the implementation can be checked immediately without updating a high-level description.

The paper's second contribution is demonstrating that CMC works well on real code by applying it to three implementations of the Ad-hoc On-demand Distance Vector (AODV) networking protocol [7]. We found 34 distinct errors (roughly one bug per 328 lines of code), including a bug in the AODV specification itself. Given our experience building systems, it appears that the approach will work well in other contexts, and especially well for other networking protocols.

[*]Supported by GSRC/MARCO Grant No:SA3276JB

[†]Supported under a National Science Foundation Graduate Research Fellowship

1 Introduction

Complex systems have complex errors. Real systems have a variety of mishandled corner cases triggered by intricate sequences of events. In practice, this leaves a residue of errors that cause system crashes but only after days or weeks of continuous execution. When detected, such problems are often very difficult to diagnose because the errors are not reproducible and the sequence of events leading to them cannot be reconstructed.

Formal verification methods are a possible way to find and diagnose such deep errors [23, 24, 29]. One option is explicit model checking, which systematically enumerates the possible states of the system. A basic model checker starts from an initial state and recursively generates successive system states by executing the nondeterministic events of the system. States are stored in a hash table to ensure that each state is explored at most once. This process continues either until the whole state space is explored, or until the model checker runs out of resources. When it works, this style of state graph exploration can achieve the effect of impractically massive testing by avoiding the redundancy that would occur in conventional testing.

Conventional model checkers usually assume that the design is described at a high level that abstracts away many details of the actual implementation. Verifying actual code using such a tool requires reconstructing this abstract description from the code. This process requires a great deal of manual effort, hampering the use of model checking in actual system design. Moreover, human errors in the manual abstraction process result in missing bugs and cause false alarms during verification, further increasing the cost and reducing the usefulness of model checking. Such errors can be introduced both when constructing the model and as a result of "drift" as the actual system evolves [5]. For these reasons, it is a notable curiosity when software is model checked, rather than an everyday occurrence.

We introduce CMC (C Model Checker) to address some of these issues. CMC works on unmodified C or C++ implementations and explores large state spaces efficiently by storing states. Like traditional model checkers, CMC achieves the equivalent of executing astronomical numbers of tests in reasonable time. However, CMC does not require writing a separate high-level model of the code, nor extracting such a model from an implementation. More importantly, it finds the bugs that are actually *in* the implementation: it does not miss implementation bugs that would be omitted from a model, nor does it waste the user's time with bugs that appear in the model but *not* in the implementation.

The idea of model checking actual implementation code has been advocated in a small number of other tools. Verisoft [15], for instance, systematically executes C implementation code but does not store states. Other software model checking tools such as [3] [25] are specialized to work only with certain classes of Java programs. CMC was designed to combine effective techniques from various research efforts within the verification community and apply them to software written in C and C++, the predominant programming languages in industry.

The ultimate goal of this work is to check systems code in general, but the initial focus is on networking code. Such code naturally follows an "event-driven" execution model which makes it a good fit for model checkers. The correctness of networking protocol implementations is especially important, since they are not only at the core of many services, but also the first target of external security attacks. Unfortunately, network protocols are difficult to design, implement and test because they involve complex interactions among multiple machines across a network and deal with various network failures such as packet losses or link failures, which are difficult to control in a test environment. Model checkers excel at checking such interactions.

CMC works well on real code, as demonstrated by the results of applying it to three implementations of the AODV networking protocol [7]. The first implementation, *mad-hoc* [21], was released two years ago and has since been under active development. The second implementation, *Kernel AODV* [20], derives from the *mad-hoc* implementation and was released less than a year ago. The third implementation, *AODV-UU* [13] was released a year ago. The AODV specification is also in active development: the first version came out in 1997 and has subsequently undergone ten revisions. While it is difficult to measure quality absolutely, one measure is that there is a formal group devoted to testing AODV implementations that has used their testbed to check the *mad-hoc* and the *AODV-UU* implementations [12].

CMC found 34 unique errors in total (as of the date of this publication), a rate of roughly one bug per 328 lines of code. Several bugs were non-trivial ones, difficult to find by any other method. In an ironic twist, model checking the *implementation* found a bug in the *specification* of AODV itself (this last error was confirmed by the authors of the AODV specification [26]). Many protocol implementations are similar to that of AODV, and CMC has enhancements that will broaden its applicability to other concurrent systems. Hence, there is good reason to believe that CMC will be useful for many systems that are difficult to debug by any other means.

2 Model Checking Overview

Fundamentally, explicit state model checking is a systematic search for error states in a state graph, which represents the behavior of some system. It is usually best to generate the graph *on the fly* so that the search can find and report errors even if the state graph is too large to search completely. This is especially important since the state graphs of systems with errors are often much larger than those of correct systems. As with most search algorithms, newly discovered states are stored in a queue. According to some policy (e.g., depth-first, breadth-first, or best-first), states are removed from this queue and the successors generated are expanded and themselves enqueued (there are usually multiple successors because of nondeterminism in the system). States that have already been searched are stored in a hash table so that their successors are not expanded more than once.

Model checking is sometimes used to prove that a system satisfies a specified property. However, it is usually more practical to use it as a bug-finding method. When model checking is applicable, it can be more effective than conventional testing in discovering bugs because of its thoroughness at exploring the state space of the system, including corner cases that might otherwise be overlooked. Model checking can be more efficient than random testing because the former searches each state at most once.

Given some code to model check, it is necessary to model the environment (e.g., the relevant aspects of the network and operating system calls). The environment model is necessary to avoid false error reports resulting from illegal inputs or state changes that would never occur in actual system execution. Many parts of the environment model would be necessary even for unit testing.

Even after the system to be checked is put into a model checker, one of the most apparent problems with model checking is that a relatively small system description can result in a huge state graph. This is called the *state explosion problem.* It has been addressed in many ways, including various methods of suppressing details of the input description (abstraction) and various optimizations to save time and, even more importantly, space. Nevertheless, the state explosion problem remains a serious difficulty in most applications of model checking. (Note that without state pruning, randomized testing will typically fare significantly worse in such situations.)

By addressing the issues described above, this paper presents an approach to pragmatically apply model checking to actual implementation code (in C or C++) to find bugs. To this end, we implemented a tool called CMC that was used to find bugs in network protocol implementations, as described in the following sections.

3 Design of CMC

CMC is a model checker that generates the state space of a given system by directly executing its C or C++ implementation. This section describes the design of CMC, beginning with a description of the tool's infrastructure. The steps required to set up a system for checking are described and illustrated with an example. The actual model checking algorithm then follows. Finally, some techniques to cope with the *state explosion problem* are discussed.

3.1 CMC Infrastructure

CMC models a system as a collection of interacting concurrent agents called *processes.* Each process runs unmodified C or C++ code from the implementation, but the CMC model checker is responsible for scheduling and executing the processes of the system being checked. CMC along with all the processes of the system run as a single operating system process. Unlike an operating system, however, CMC tries to search many *possible* system states that can be reached by alternative scheduling decisions and other nondeterministic events. To search these different possibilities, CMC must be able to save and restore the complete state of the modelled system.

Every process of the system executes in its own heap and stack. At any instant, the state of a process consists of a copy of its global (and static) variables, heap, stack and its context registers. The processes communicate with each other through a shared memory that is accessible in the context of all processes. The state of the system is defined as the union of the states of all processes along with the contents of the shared memory.

Once scheduled, a process is allowed to execute a deterministic and non-blocking set of instructions defined as a *transition.* A transition is an atomic step the system can take and determines the degree of interleaving among processes. The protocols to which CMC has been applied follow an *event-driven* execution model, where a set of event handler routines process incoming events such as packet arrivals and timeouts. In an event-driven protocol, each event handler can be mapped to a transition in CMC. Moreover, as each event handler preserves the state of the stack and registers, only the global variables and the heap need to be saved and restored.

Many protocols are written in an event-driven style, so the event-driven model may not be as restrictive as it may seem. However, it is feasible to save and restore full states (including the stack) of modelled processes as well. This feature has been implemented and is currently being evaluated, but was not necessary for the results reported in this paper.

3.2 Creating a CMC Model from the Implementation

Figure 1 shows a skeleton event-driven implementation of a routing protocol, very similar to the AODV protocol that was actually checked (see section 4). This implementation is used as a running example in the following discussion. The `main()` function of the implementation calls the initialization function (line 34), and then enters an event dispatch loop (line 35). Depending on the input event, it calls one of four event handlers defined in lines 1 through 26. Each handler processes an event: a user request for a route to a destination, a request from another node, a response from another node to one of its previous requests, and a timer event requiring the protocol to invalidate its old routes.

This subsection describes the three steps that the user must perform to apply CMC to a protocol. Steps 1 and 2 essentially provide a unit test scaffold, which would be required for most test environments, such as running the implementation in a simulator. Step 3 is the only additional requirement for CMC.

Step 1: Specifying Correctness Properties. Before any system can be tested, the user must specify some correctness properties. Some properties are domain independent; for example, the program should not access illegal memory and should not leak memory. Some domain-specific properties can be specified as assertions at particular points in the implementation (e.g., the example protocol should never return an invalid route in line 4). In many cases, a careful implementer will have placed these assertions in the code long before CMC is applied. Other properties are inherently global, such as the requirement that there are no loops in the routing table. Such properties are specified as Boolean functions (written in C) that access the datastructures of process contexts.

Step 2: Specifying the Environment. Next, the user must build a test environment that adequately represents the behavior of the actual environment in which the protocol is executed. For networking protocols, the environment model fakes an operating system and anything outside of the protocol that is necessary for it to function. (The decision as to which part of the system is to be checked and which is the environment is decided by the user.) The environment model is a collection of substitute API functions and data structures to emulate its state. The environment should be modelled in as little detail as possible – otherwise, many superfluous states will be generated to model environmental behavior irrelevant to checking the protocol.

Many functions can be replaced by simple "stubs." For example, `gettimeofday()` might return a constant or contain a counter. In the example, the model requires a network for exchanging routing packets. A simple network could be modelled as an unordered queue of bounded length. The model would include its own versions of interface functions, such as the `broadcast_request()` function that sends packets to the network. The environment may also contain several processes. For ex-

```
 1 /* event handlers */
 2 on_user_request (dest_ip) {
 3   if(route_table has a route for dest_ip)
 4     return route for dest_ip
 5   else
 6     broadcast_request(dest_ip)
 7 }
 8
 9 on_recv_request (dest_ip) {
10   if(route_table has a route for dest_ip)
11     send_response (route for dest_ip)
12   else
13      broadcast_request(dest_ip)
14 }
15
16 on_recv_response (route) {
17   install route in route_table
18   if(route needs to be forwarded)
19     send_response (route)
20 }
21
22 on_timeout(){
23   for each route in route_table
24     if(route too old)
25       remove route from route_table
26 }
27
28 init(){
29   route_table = null
30   insert self route for my_ip
31 }
32
33 main(){
34   init()
35   while(1){
36     /* event dispatch loop */
37     depending on event, call one of
38        on_user_request(...)
39        on_recv_request(...)
40        on_recv_response(...)
41        on_timeout(...)
42   }
43 }
44
```

Figure 1: A simple routing protocol implementation.

```
void* malloc(size_t n){
  if(CMCChoose(2) == 0)
     return 0; //nondeterministic failure

  // alloc n bytes from heap and return
}
```

Figure 2: Implementation of `malloc()` in CMC. `malloc()` nondeterministically fails to allocate memory.

ample, a process that nondeterministically removes a packet from the network can be used to model a lossy network.

To represent nondeterminism in the environment, CMC provides a `CMCChoose()` function (similar to VS_toss in Verisoft[15]). CMCChoose takes an integer argument n and returns an integer in the range $(0 \ldots n-1)$. CMCChoose arbitrarily selects one out of the n possibilities in the environment. An example, shown in Figure 2, is the `malloc()` implementation that can either allocate the requested memory from the CMC heap or fail by returning NULL. CMCChoose is used to make one of these two choices. CMC will attempt to try all possible return values for each call to CMCChoose. Since calls to CMCChoose appear in the environment code or in implementations of standard system functions (such as `malloc() and select()`), it is generally not necessary to modify the actual implementation.

Providing an environment model can be time consuming. It is therefore important to reduce the modelling effort required to apply CMC to a previously unchecked protocol. A first obvious step is to engineer the models so that they are as re-usable as possible, reducing the incremental effort of checking a new protocol. This is especially beneficial when related protocols are checked (which is a large part of the reason this paper checks three different implementations of the same protocol). Finding other ways to reduce the cost of environmental modelling is an interesting area for future work.

Step 3: Identifying the Initialization Functions and Event Handlers. Given an event driven system, the user should provide the initialization functions and the event handlers for each process in the system.

The user should also provide a *guard function* for each event handler, which is a Boolean function that determines when that event handler is enabled in a given state. For instance, the guard function for the on_`recv_request()` handler returns true only if a request is pending for this particular process in the network.

3.3 CMC Model Checking Algorithm

Given a model of the system built as described above, CMC explores the state space of the system by executing various traces of interleaving transitions. The pseudocode for the algorithm is shown in Figure 3. The algorithm maintains two data structures: a hash table of states seen during the search, and a queue of states seen but whose successors are yet to be generated. The hash table guarantees that the algorithm explores the subgraph rooted at a state at most once.

3.3.1 Generating the Initial State

CMC computes the initial state as follows. Starting from a copy of the global variables (as initialized by the linker), CMC calls the initialization function for each process. The initial state consists of the states of the processes immediately after their initialization functions have been called, along with the values of the initialized shared memory.

3.3.2 Generating Successor States

To generate the state graph on-the-fly, CMC needs to be able to compute the set of possible immediate successors of a state. Each state in the state space of the system may have several successors because of nondeterminism, which arises from several sources: the choice of which process to execute, the choice of which enabled transition within the process to execute, and nondeterministic values returned by calls to CMCChoose.

From a given state, CMC chooses a process and one of its enabled event handlers to schedule. Next, CMC restores the context of the process by copying the contents of the heap and the global variables of the process from the state. The event handler is then called. This function eventually returns because it is guaranteed to be atomic. At this point, the context of the process state is saved, yielding a new system state. CMC generates all successors of a state by repeating the above process for all nondeterministic choices.

3.3.3 Checking Correctness Properties

During model checking, CMC checks for a range of correctness properties, from simple pointer access

```
void modelCheck(){
   SystemState{
      sharedMem;   // Contains Network
      procState[N];// N processes
   } initial, current, successor;

   Queue StateQ;
   Hash VisitedStates;

  // Build initial state.
  forall processes (0 <= pid < N){
    call pid's initFn();
    initial.procState[pid] = saveCtxt();
  }
  call sharedMem's initFn();
  initial.sharedMem = getSharedMem();

  StateQ.insert(initial);
  while(current = StateQ.pop()) {
    VisitedStates.add(current);

    //Repeat forall nondeterministic choices
    forall processes (0 <= pid < N)
    forall event handlers e of pid
    forall return values of CMCChoose calls{

      // set proc context and sharedMem
      restoreCtxt(current.procState[pid]);
      setSharedMem(current.sharedMem);

      if(e is not enabled)
         continue;

      e(); // Call event handler

      // Construct next state
      successor.sharedMem = getsharedMem();
      successor.procState[pid] = saveCtxt();
      forall processes with pid' != pid{
        // the state did not change
        successor.procState[pid']
          = current.procState[pid'];
      }
      if(successor in VisitedStates)
        continue;
      if(successor fails assertions)
        generate error
      StateQ.insert(successor);
    }
  }
}
```

Figure 3: Pseudocode for the CMC model checking algorithm.

violation errors to complex protocol bugs.

During the execution of an event handler, CMC runs the implementation code directly, automatically catching errors such as pointer access violations and program assertion failures present in the code. In addition, CMC detects use-after-free bugs by overwriting any freed memory with a random value.

Once a state is generated, CMC checks for violations of user-provided system invariants (such as the absence of global routing loops). Also, CMC detects memory leaks in each generated state. While this can be achieved by a standard mark-and-sweep algorithm to find all reachable memory, such an algorithm is not yet implemented in CMC. Instead, in our case study, CMC detects memory leaks as follows: starting from a copy of the current state, CMC calls various cleanup functions present in the implementation itself. Any heap memory that is left allocated is reported as leaked. This approach, while requiring additional manual effort, can also potentially find bugs in the cleanup code.

In the future, the CMC approach could easily be coupled with other dynamic debugging tools such as Purify[27] or StackGuard[6]. These tools can catch run-time errors such as uses of uninitialized memory, stack overflows, etc. Such tools would be more effective when used with CMC than with ordinary testing, because CMC would achieve greater effective test coverage for a given level of user effort than conventional software testing methods.

3.4 Handling State Space Explosion

One of the most serious problems with model checking in practice is the so-called "state explosion problem." The state space of a system can be very large, or even infinite. Thus, at the outset, it is impossible to explore the *entire* state space with limited resources of time and memory. However, CMC provides various techniques to search the state space efficiently before running out of resources. Though unable to formally prove the correctness of the implementation, CMC is able to catch a wide range of errors, including errors involving intricate interactions among multiple processes.

For model checkers, memory is more critical a resource than time. During model checking, most of the memory is consumed by the hash table containing the states visited and the queue of states whose successors are yet to be generated.

CMC uses *hash compaction* [28] to reduce the memory requirements in the hash table by several orders of magnitude. For each state, CMC computes a small signature (usually four to eight bytes). Instead of storing the entire state, which can be on the order of kilobytes, its signature is stored in the hash table. Compacting states can lead to conflicts in the hash table where two different states compute to the same signature. However, for state spaces on the order of hundred million states with practical hash table sizes of several hundred megabytes, the probability of missing even a single state due to a signature conflict can be reduced to 0.1% or lower [28].

The states in the queue cannot be compacted because all the information in them is needed to compute successor states. However, the queue has good locality of reference, so much of it can be swapped to disk during model checking. Moreover, successive states in the queue usually have a lot of commonality and can thus be compressed. For instance, every transition in CMC changes at most one process state; therefore, it is sufficient to store only this difference when generating a successor state.

Standardizing Data Structures: CMC, by default, interprets states as streams of bits. However, two equivalent data structures in memory might have different representations. For example, if two states differ only in the order in which objects were allocated on the heap, they should be considered effectively the same. CMC can automatically transform states by deterministically traversing pointer data structures, arranging objects in the heap by the order they are visited. The signature for the transformed state can then be saved in the state table. This process could be performed simultaneously with the mark-and-sweep algorithm used to detect memory leaks. A mostly automatic tool for this traversal is under development using the MC framework [11]. For the case study discussed in Section 5, the traversal code was written manually.

There may be additional equivalences between states that depend on the particular use of data structures in a program. For example, when an implementation uses a linked list to store an *unordered* collection of objects, the behavior of the implementation is independent of the order of objects in the list. In this case, the user can provide a function to sort the list before the automatic standardization transformations are applied.

Finally, some of the most effective reductions in the state space are achieved through methods that risk missing some errors for the benefit of catching the remaining ones more efficiently.

Down-scaling: One obvious approach is to reduce the scale of the system being described [10]. In figure 1, for instance, the model might restrict the number of routing nodes in the network to, say, three or four. Hard-to-find bugs usually involve complex interactions among a *small* number of processes, and are therefore preserved even after down-scaling. Of course, this may miss bugs that only occur for larger instances of the system.

Abstraction of States: In addition to standardizing distinct but equivalent states, it is also possible to eliminate information that the user judges to be unimportant for the properties checked. This abstraction process is done by ignoring certain memory locations when computing the hash signature of the state. By abstracting states, it is possible to miss errors. However, as the abstraction is done during the hash computation, and not on the actual (concrete) state, this does *not* produce any false positives.

Heuristics: When exhaustive checking of the entire state space is infeasible and all else fails, CMC can act as an automated testing framework whereby a large number of scenarios can be checked intelligently. The mere fact that CMC is able to cache states already prevents redundant simulations. The goal, however, is to exercise as many interesting scenarios as possible before memory is exhausted.

To that end, we have done some preliminary work in using heuristics to prioritize the state space search. The first class of heuristics involves dropping states altogether if they are deemed uninteresting. The second class of heuristics involves exploring more interesting states first using best-first search. CMC contains a module to monitor state variables to keep a history of which state bits have changed during checking. The basic idea is that if the number of bit positions that have changed since the initial state suddenly increases or if variables take on less frequented values, the state is considered more interesting and explored earlier. This heuristic tends to bias the search toward cases where outliers occur or where states seem to diverge from the norm. This idea was adapted from DIDUCE [17], a tool that flags such divergent cases and reports them to the user during program testing.

Preliminary results indicate that all the errors discovered with the use of the heuristics could be discovered with simple depth-first search. But, the use of heuristics often accelerated the discovery of errors and produced shorter examples of executions leading to a given error. However, much more experimentation with various heuristics is needed on a wider range of protocols to arrive at reliable conclusions.

The next three sections describe the application and results of using CMC to check three AODV protocol implementations.

4 Description of the AODV Protocol

AODV (Ad-hoc On-demand Distance Vector)[7] is a loop-free routing protocol for ad-hoc networks. It is designed to be self-starting in an environment of mobile nodes, withstanding a variety of network behaviors such as node mobility, link failures and packet losses. This section describes the AODV protocol in brief; the reader is referred to [7] for complete details of the protocol.

At each node, AODV maintains a routing table. The routing table entry for a destination contains three essential fields: a next hop node, a sequence number and a hop count. All packets destined to the destination are sent to the next hop node. The sequence number acts as a form of time-stamping, and is a measure of the freshness of a route. The hop count represents the current distance to the destination node.

Suppose we have two nodes a and b such that b is the next hop of a to some destination d. Also, suppose the sequence number and hop count of the routes to d at a and b are $(seq_a, hcnt_a)$ and $(seq_b, hcnt_b)$ respectively. Then the AODV protocol maintains the following property at all times:

$$(seq_a < seq_b) \vee (seq_a = seq_b \wedge hcnt_a > hcnt_b)$$

In other words, b either has a newer route to d than a, or b has a shorter route that is equally recent. Under this partial order constraint, the protocol is guaranteed to be free of routing loops [2].

In AODV, nodes discover routes in request-response cycles. A node requests a route to a destination by broadcasting an *RREQ* message to all its neighbors. When a node receives an RREQ message but does not have a route to the requested destination, it in turn broadcasts the RREQ message. Also, it remembers a *reverse-route* to the requesting node which can be used to forward subsequent responses to this RREQ. This process repeats until the RREQ reaches a node that has a valid route to the destination. This node (which can be the destination itself) responds with an *RREP* message. This RREP is unicast along the reverse-routes of the intermediate nodes until it reaches the original requesting node. Thus, at the end of this request-response cycle a *bidirectional* route is established between the requesting node and the destination. When a node loses connectivity to its next hop, the node invalidates its route by sending an *RERR* to all nodes that potentially received its *RREP*.

On receipt of the three AODV messages: RREQ, RREP and RERR, the nodes update the next hop, sequence number and the hop counts of their routes in such a way as to satisfy the partial order constraint mentioned above [7].

5 The AODV model

This section describes the AODV model for three implementations of the AODV protocol: *mad-hoc* (Version 1.0) [21], *Kernel AODV* (Version 1.5) [20], and *AODV-UU* (Version 0.5) [13]. The mad-hoc implementation runs as a user space daemon and contains approximately 5500 lines of code. The Kernel AODV implementation is built by NIST and is based on the mad-hoc implementation. It contains 7500 lines of code and runs as a loadable kernel module in Linux and ARM based PDAs. The AODV-UU implementation runs as a user space daemon on Linux and has been ported to the ns-2 [22] simulator. It contains roughly 7700 lines of code.

The AODV model was reused with minor modifications for all three implementations. The model is built as follows:

Correctness Properties: Table 1 lists the correctness properties checked by the AODV model. Apart from the generic assertions checked by CMC, the model contains a global invariant that checks for routing loops. The model also performs sanity checks on the routing table entries and the network messages, such as range violations of the fields.

The Environment: The environment of the model consists of a network modelled as a bounded-length, unordered message queue. The model simulates a message loss by nondeterministically dequeuing a message. The message queue is shared by all of the nodes and thus models a completely connected topology.

The implementations use a wrapper function to send network packets. The model provides an alternate definition to the wrapper function to copy packets to the network model. Additionally, for the Kernel AODV implementation, the model provides implementations for twenty-two kernel functions (such as

Types of Checks	**Examples**
Generic Assertions	Segmentation violations, memory leaks, dangling pointers.
Routing Loop Invariant	The routing tables of all nodes do not form a routing loop.
Assertions on Routing Table Entries	At most one routing table entry per destination. No route to self in the AODV-UU implementation. The hop count of the route to self is 0, if present. The hop count is either infinity or less than the number of nodes in the network.
Assertions on Message Fields	All reserved fields are set to 0. The hop count in the packet can not be infinity.

Table 1: Properties checked in AODV.

Enabling Condition	**Event**
Invalid or no route to destination	Initiation of route request
Pending message in the network	Receipt of AODV message
Pending message in the network	Message loss
Valid route in the routing table	Timeout of a route
Always enabled	Detection of link failure
Always enabled	Node reboot

Table 2: The set of event handlers used in AODV model checking.

kmalloc and printk) and a user space version of the socket buffer library.

Initialization Functions and the Event Handlers: All three implementations have an event dispatch loop that calls various event handlers. The initialization functions of the model are obtained by executing the code before the event dispatch loop. The model maps every event handler called from the dispatch loop to a transition. The model simulates a node reboot by calling the initialization function which implicitly resets the contents of the routing table. The list of transitions and their respective enabling conditions is shown in Table 2.

Table 3 shows the lines of code from the three implementations executed within our framework against the lines of code for the model itself. The correctness specifications are mostly shared by the three implementations. AODV-UU uses a different representation of the routing table and thus required additional correctness specifications. The network model of the environment is shared by all implementations.

Dealing with State Space Explosion

The state space of the AODV protocol is essentially infinite. The protocol allows an arbitrary number of nodes in a network. Also, each node has two types of unbounded counters, a *sequence number* to measure the freshness of a route and a *broadcast id* that is incremented by a node on each broadcast. To do any effective search in such an infinite state space, it is necessary to bound the search. In our experiments, we downscaled the AODV model to run with 2 to 4 processes. The model discarded any state in which the sequence numbers or the broadcast ids exceeded a predefined limit. Also, the size of the message queue in the network was bounded to sizes of 1 to 3. These processes may cause CMC to miss errors. However, even after applying such bounds, the remaining state space contained enough interesting behavior to uncover numerous bugs (Section 6).

Time values stored in the state are another source of state space explosion. For instance, every route response (RREP) contains a lifetime field that determines the freshness of the route. On receipt of this packet, a node adds the lifetime to the current clock value to determine the time at which the route becomes stale. This absolute value is stored in the routing table and can thus increase the state space size. The AODV model gets around this problem by modelling route timeouts as nondeterministic events and setting all time variables to predefined constants. Also, the environment of the model contains a definition of the `gettimeofday()` function that always returns a constant value. The handling of time in the model can miss timing related errors

Protocol	Checked Code	Correctness Specification	Environment			State Canonicalization
			network	stubs	*skbuff*	
mad-hoc	3336	301	400	100	-	165
Kernel AODV	4508	301	400	266	1210	179
AODV-UU	5286	332	400	128	-	185

Table 3: Lines of implementation code vs. CMC modelling code.

and can potentially lead to false positives when an error reported can be caused by a sequence of timeouts that is impossible in the real protocol.

Also, the AODV model contains hand-written code to traverse the routing table (implemented as a linked list in the mad-hoc and Kernel AODV implementations, and as a hash table in the AODV-UU implementation). This traversal code created a canonicalized representation of the routing table, which along with the global variables formed the state of an AODV node of the model. The amount of lines required for this traversal code is shown in the last column of Table 3.

6 Results

Table 4 summarizes the set of bugs found using CMC in the three AODV implementations. The bugs range from simple memory errors to protocol invariant violations. We found a total of 40 bugs of which 34 were unique. The Kernel AODV implementation has 5 bugs (shown in parenthesis in the table) that are instances of the same bug in mad-hoc. Also, the AODV specification bug causes a routing loop in all three implementations.

Currently, CMC stops after finding the first bug in the model. It prints the failed assertion and a trace of events starting from the initial state to the error state. After a bug is fixed, CMC is run again to find bugs iteratively. Most bugs were found within minutes of model checking time; the longest took roughly 40 minutes.

We describe the bugs below at a high level to give a feel for the breadth of coverage and focus on four of the more interesting bugs to give a feel for its depth.

Memory errors. The first three error classes illustrate the mishandling of dynamically allocated memory: not checking for allocation failure (12 errors), not freeing allocated memory (8 errors), and using memory after freeing it (2 errors).

All implementations checked that the pointer returned by `malloc` was not null. However, functions that call `malloc` can also indirectly return null pointers when allocations fail. The code only erratically checked such cases. Since CMC directly executes the implementation, such errors were manifested as segmentation faults.

Most of the memory-related bugs were straightforward. However, there were several interesting errors where the code would correctly check for allocation failure, but its recovery code was broken. Figure 4 gives a representative error. Here, the code attempts to allocate `rerrhdr_msg.dst_cnt` temporary message buffers. It correctly checks for `malloc` failure and breaks out of the loop. However, the code after the loop assumes that `rerrhdr_msg.dst_cnt` list entries were indeed allocated. This assumption leads to two bugs. The first (intraprocedural) error

```
/*aodv_deamon.c:aodv_recv_message:*/
...
for(rerri=0; rerri<rerrhdr_msg.dst_cnt;rerri++)
{
   if (!(tp = malloc(sizeof(*tp))))
      break; /* Skip to next packet */
   tp->next = rerrhdr_msg.unr_dst;
   rerrhdr_msg.unr_dst = tp;
   ...
}
// BUG: assumes rerrhdr_msg.dst_cnt buffers
// were allocated!
rec_rerr(info_msg, &rerrhdr_msg);

// Free the list of structs sent to rec_rerr()
for(rerri=0; rerri<rerrhdr_msg.dst_cnt;rerri++)
{
  // BUG: Can be NULL if malloc failed above!
  tp = rerrhdr_msg.unr_dst;
  rerrhdr_msg.unr_dst=rerrhdr_msg.unr_dst->next;
  free(tp);
}
```

Figure 4: Mishandled `malloc` failure: if `malloc` fails, the loop will exit after allocating less than `rerrhdr_msg.dst_cnt` buffers. The two errors are in code that assumes `rerrhdr_msg.dst_cnt` buffers were allocated. Both lead to segmentation faults.

	mad-hoc	Kernel AODV	AODV-UU
Mishandling `malloc` failures	4	6	2
Memory Leaks	5	3	0
Use after free	1	1	0
Invalid Routing Table Entry	0	0	1
Unexpected Message	2	0	0
Generating Invalid Packets	3	2 (2)	1
Program Assertion Failures	1	1 (1)	1
Routing Loops	2	3 (2)	1 (1)
Total	18	16 (5)	6 (1)

Table 4: Number of bugs of each type in the three implementations of AODV. The figures in parenthesis show the number of bugs that are instances of the same bug in the mad-hoc implementation.

attempts to dequeue **`rerrhdr_msg.dst_cnt`** buffers off of the **`rerrhdr_msg.unr_dst`** list in order to free them. Since the list has fewer entries than expected, the code will attempt to use a null pointer and get a segmentation fault. The second (interprocedural) error, in **`rec_rerr`**, similarly tries to walk over the **`rerrhdr_msg.dst_cnt`** list entries and seg faults because the list is too short.

Most of the memory leaks were similarly caused by mishandled allocation failures. Commonly, code would attempt to do two memory allocations and, if the first allocation succeeded but the second failed, would return with an error, leaking the first pointer.

Unexpected messages. CMC detected two places where unexpected messages would cause mad-hoc to crash with a segmentation violation. Figure 5(a) shows one of the errors. The error happens because AODV encodes state in its messages. In this error:

1. The current node n receives a Route Request (RREQ) message from node *req* requesting a route to node *dst*.
2. Node n inserts a *reverse route* to *req* in its routing table.
3. Then, n looks up the route to *dst* in its routing table.
4. If the route is not there, n re-broadcasts the RREQ message. The RREQ message contains the IP address of both the destination node *dst* and the requesting node *req*.
5. The response to this request, a Route Response (RREP) message, includes both the route to *dst* and the IP address of *req*.
6. Node n inserts the new route to *dst* in its routing table. It then attempts to relay this route to *req* by looking up the route to *req*. In the normal case, this lookup will return the reverse route inserted in Step 2.

This last step causes the error. The code assumes the normal case and uses the result of the routing table lookup for *req* without checking for null. However, the lookup could fail for two reasons. First, if the machine has rebooted, the implementation will start with an empty routing table. If an old RREP message arrives after the reboot, the lookup of *req* will return a null pointer. Second, an attacker could send a bogus RREP with a node address that does not exist, crashing the router.

Invalid messages. There were 4 cases of invalid packets being created, 2 cases of using uninitialized variables (these could not be detected by gcc -Wall), and 2 cases where invalid routes were used to send routing updates, violating the AODV specification (Figure 5(b) gives a representative example). CMC also detected 2 instances of integer overflow which resulted in program assertion failures. The implementations use an 8 bit integer to store the hop counts and use 255 to represent a hopcount of infinity. In these error cases, an infinite hopcount was erroneously incremented to 0.

Routing loops. CMC found three routing loops. Two of these bugs are caused by implementation errors. The third routing loop is due to an error in the AODV protocol specification.

The first routing loop is caused when the implementation fails to increment a sequence number while processing specific RERR messages.

Another loop is caused when the implementation performs a sequence number comparison before a subsequent increment, while the AODV specification requires the comparison to be done after the increment.

```
/* madhoc:rrep.c:rec_rrep */
...
/* If I'm not the destination of the RREP
   I forward it */
if(my_rrep->src_ip != my_info->ip_pkt_my_ip) {
  ...
  // Get the entry to the source from RT.
  rt_src = getentry(my_rrep->src_ip);

  // BUG: rt_src may not exist!
  if (add_precursor(rt_src, rt->nxt_hop) == -1)
  ...
  // Send gratuitous RREP to destination
  // BUG: rt_src can be invalid
  // (i.e rt_src->hop_cnt == 255 )
  // must check after getentry.
  my_rrep.hop_cnt = rt_src->hop_cnt;
  if (send_datagram(my_info, &my_rrep,
                    sizeof(my_rrep)) == -1)
    ...
```

Figure 5: Two bugs: an unexpected message and an invalid route response. (a) An unexpected route-response (RREP) message causes **getentry** to return null, crashing the machine. (b) If a route returned by **getentry** has been invalidated the hop-count will be 255. However, the code does not check for this and sends the message.

The specification bug. This bug involved the handling of RERR ("route error") messages. When a node receives an RERR from its next hop, it sets the sequence number of its route to the sequence number in an RERR message. Under normal conditions this is the right thing to do. However, when the underlying link layer can reorder messages, the RERR message might have an outdated sequence number resulting in the node setting its sequence number to an older version. This can ultimately result in a routing loop. This bug was mentioned to the authors of the protocol with a suggested fix. Both the bug and the fix were accepted by the protocol authors[26]. Figure 6 gives both the error and the fix.

The specification bug was found by running 4 AODV nodes using a depth-first search of the state space. CMC came up with an error trace of length 93. Using best-first search, it was possible to find traces as short as 27. Performing a breadth-first search of the state space would give the shortest trace. However, breadth-first search on AODV ran out of resources without finding the bug. A carefully hand-crafted simulation of the bug required at least 20 transitions. Such a complex error would be very difficult to catch using conventional means of testing.

```
/* madhoc:rerr.c:rec_rerr */
...
// Get pointer to route table for destination IP
tmp_rtentry = getentry(tmp_unr_dst->unr_dst_ip);
if(tmp_rtentry != NULL && ...) {
  // BUG: uses sequence number from incoming
  // message in tmp_unr_dst without validation.
  // Should check:
  //   if(tmp_rtentry->dst_seq >=
  //          tmp_unr_dst->unr_dst_seq)
  //      return -1;
  tmp_rtentry->dst_seq = tmp_unr_dst->unr_dst_seq;
```

Figure 6: The specification bug: the sequence number from an incoming message is used without validation, causing "time" to go backwards when messages are reordered. Fortunately, while the error was not obvious (surviving 6 rounds of specification revisions) the fix is trivial.

7 Related Work

This paper proposes an initial approach to systematically and efficiently verify a large class of C and C++ software without having to create abstract models in a different language. The following compares our work using CMC to other efforts in traditional model checking, software model checking, and static analysis.

Traditional Model Checking: The basic idea of using state graph search to verify network and communication protocols is quite old, dating back to at least 1978 [16, 30]. In recent decades, model checking has made significant progress in tackling the verification of complex, concurrent systems. Tools such as SMV[19], SPIN[18], and Murphi[10] have been used to verify hardware and software protocols by exhaustively searching the state space. By caching states and employing sound state reduction techniques, these tools can detect non-trivial bugs.

The drawback of traditional model checkers is that the system to be verified must be modeled in a particular description language, requiring a significant amount of manual effort that can easily be error prone. CMC was specifically designed with the goal of reducing the amount of work that is required to go from software development to systematic verification.

Software Model Checking: Some recent formal verification tools have already used the idea of executing and checking systems at the implementation level. Verisoft [15], for instance, systematically executes and verifies actual code and has been used to successfully check communication protocols written in C.

However, Verisoft does not store states and can thus potentially explore a state more than once. This problem is alleviated to some degree by partial order reduction, a sound state space reduction technique implemented in Verisoft that eliminates the exploration of redundant interleavings of transitions created by commutative operations. Nevertheless, this technique requires hints to be provided by the user and/or some static analysis of the code to determine dependencies between transitions; indeed, when the set of possible transitions in a system have a high degree of interdependence, as is the case with the handlers in the protocol code we verified, partial order methods become less effective. Finally, interesting systems almost always have state spaces with cycles and in such cases Verisoft is limited to checking only up to a fixed depth.

Java PathFinder [3] uses model checking to verify concurrent Java programs for deadlock and assertion failures. It relies on a specialized virtual machine that is tailored to automatically extract the current state of a Java program. Much like CMC, Java PathFinder compresses and stores states in a table to prevent redundant searches and relies on various abstraction techniques to curb the state space explosion problem. The infrastructure on which JPF relies, however, can not be applied to software written in C or C++, which are still the predominant languages used in system software development.

SLAM [1] is a tool that converts C code into abstracted skeletons that contain only Boolean types. SLAM then model checks the abstracted program to see if an error state is reachable. One difficulty in using a tool like SLAM is giving a specification of the correct behavior of the system. Because SLAM is a static tool, writing a specification that "no routing loops are possible" would be difficult because it depends on the interleaved event behavior of multiple nodes. Furthermore, SLAM does not deal with concurrent environments that contain multiple processes, queues, etc.

Static Analysis: Static analysis has also gained ground in recent years in detecting bugs in software. Tools such as ESC [9], LCLint [14], ESP [8], and the MC Checker [11] have been used to check source code for errors that can be statically detected with minimal manual effort. While static techniques are good for finding a specific set of errors, the CMC approach can find deep conceptual errors in the code such as emergent routing loops that are difficult to find statically. In addition, CMC does not suffer from too many false positives since every scenario checked is a valid execution path.

8 Conclusion and Future Work

This paper has described CMC, a model checker targetting subtle bugs in systems code, and experimental results from using CMC to check three implementations of the AODV routing protocol. The key features of CMC are that it checks implementation code directly and stores states to avoid redundant state explorations. Initial experiences with CMC are very encouraging: CMC is powerful enough to discover non-trivial bugs both in the implementation and the specification of protocols.

We are currently using CMC to verify larger and more complex protocols. For wider use, it is essential to automate the process of converting an implementation of a system to its CMC model as much as possible. While the results reported here did require considerable manual effort, future improvements to CMC should significantly reduce this.

We are also exploring the use of heuristics to efficiently search the state space. Our initial findings suggest that simple heuristics provide huge improvements in the state space search. For instance, we have implemented a monitor that detects counters and other "rogue" variables (such as uninitialized variables and statistics variables). This monitor abstracts away such variables from the system state, automatically pruning an otherwise infinite state space. Another interesting avenue for research is to use simple facts discovered through static analysis of the code to direct the search to interesting parts of the state space.

9 Acknowledgments

We thank Satyaki Das for thoughtful discussions on the paper. Also, we thank Miguel Castro and various anonymous reviewers for providing valuable comments and suggesting improvements on previous versions of this paper.

References

[1] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[2] K. Bhargavan, D. Obradovic, and C. Gunter. Formal verification of standards for distance vector routing protocols, 1999.

[3] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.

[4] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE 2000*, 2000.

[6] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[7] C.Perkins, E. Royer, and S. Das. Ad Hoc On Demand Distance Vector (AODV) Routing. IETF Draft, http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-10.txt, January 2002.

[8] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *Conference on Programming Language Design and Implementation*, 2002.

[9] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking, 1998.

[10] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[11] D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[12] Erik Nordstrom *et al.* Ad hoc protocol evaluation testbed. http://apetestbed.sourceforge.net/.

[13] Erik Nordstrom *et al.* AODV-UU Implementation. http://user.it.uu.se/ henrikl/aodv/.

[14] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[15] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[16] J. Hajek. Automatically verified data transfer protocols. In *Proceedings of the 4th ICCC*, pages 749–756, 1978.

[17] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[18] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[19] McMillan K. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[20] Luke Klein-Berndt and et.al. Kernel AODV Implementation. http://w3.antd.nist.gov/wctg/aodv_kernel/.

[21] F. Lilieblad and et.al. Mad-hoc AODV Implementation. http://mad-hoc.flyinglinux.net/.

[22] S. McCanne and S. Floyd. UCB/LBNL/VINT network simulator - ns (version 2), April 1999. http://www.isi.edu/nsnam/ns/.

[23] K.L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51. Tokyo, Japan Inf. Process. Soc., 1991.

[24] G. Nelson. *Techniques for program verification*. Available as Xerox PARC Research Report CSL-81-10, June, 1981, Stanford University, 1981.

[25] D. Park, U. Stern, J. Skakkebaek, and D. L. Dill. Java model checking. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.

[26] Charles E. Perkins, Elizabeth M. Royer, and Samir R. Das. Private Email Communication.

[27] Rational Software. Purify: Advanced run-time error checking for C/C++ developers. http://www.rational.com/products/purify_unix/.

[28] U. Stern and D. L. Dill. A New Scheme for Memory-Efficient Probabilistic Verification. In *IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, 1996.

[29] U. Stern and D.L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[30] C.H. West. General technique for communications protocol validation. *IBM Journal of Research and Development*, 22(4), 1978.

Practical, transparent operating system support for superpages

Juan Navarro‡ Sitaram Iyer† Peter Druschel† Alan Cox†

{jnavarro, ssiyer, druschel, alc}@cs.rice.edu

†*Rice University*

‡*Rice University and Universidad Católica de Chile*

Abstract

Most general-purpose processors provide support for memory pages of large sizes, called superpages. *Superpages enable each entry in the translation lookaside buffer (TLB) to map a large physical memory region into a virtual address space. This dramatically increases TLB coverage, reduces TLB misses, and promises performance improvements for many applications. However, supporting superpages poses several challenges to the operating system, in terms of superpage allocation and promotion tradeoffs, fragmentation control, etc. We analyze these issues, and propose the design of an effective superpage management system. We implement it in FreeBSD on the Alpha CPU, and evaluate it on real workloads and benchmarks. We obtain substantial performance benefits, often exceeding 30%; these benefits are sustained even under stressful workload scenarios.*

1 Introduction

Modern general-purpose processors provide virtual memory support, using page tables for address translation. Most processors cache virtual-to-physical-address mappings from the page tables in a translation lookaside buffer (TLB) [10]. *TLB coverage* is defined as the amount of memory accessible through these cached mappings, i.e., without incurring misses in the TLB. Over the last decade, TLB coverage has increased at a much lower pace than main memory size. For most general-purpose processors today, TLB coverage is a megabyte or less, thus representing a very small fraction of physical memory. Applications with larger working sets can incur many TLB misses and suffer from a significant performance penalty. To alleviate this problem, most modern general-purpose CPUs provide support for *superpages*.

A *superpage* is a memory page of larger size than an ordinary page (henceforth called a *base page*). They are usually available in multiple sizes, often up to several megabytes. Each superpage occupies only one entry in the TLB, so the TLB coverage dramatically increases to cover the working set of most applications. This results in performance improvements of over 30% in many cases, as we demonstrate in Section 6.2. Recent research findings on the TLB performance of modern applications state that TLB misses are becoming increasingly performance critical [9].

However, inappropriate use of large superpages can result in enlarged application footprints, leading to increased physical memory requirements and higher paging traffic. These I/O costs can easily outweigh any performance advantages obtained by avoiding TLB misses. Therefore the operating system needs to use a mixture of page sizes. The use of multiple page sizes leads to the problem of physical memory fragmentation, and decreases future opportunities for using large superpages. To ensure sustained performance, the operating system needs to control fragmentation, without penalizing system performance. The problem of effectively managing superpages thus becomes a complex, multi-dimensional optimization task. Most general-purpose operating systems either do not support superpages at all, or provide limited support [6, 19, 20].

This paper develops a general and transparent superpage management system. It balances various tradeoffs while allocating superpages, so as to achieve high and sustained performance for real workloads and negligible degradation in pathological situations. When a process allocates memory, our system reserves a larger contiguous region of physical memory in anticipation of subsequent allocations. Superpages are then created in increasing sizes as the process touches pages in this region. If the system later runs out of contiguous physical memory, it may preempt portions of unused contiguous regions from the processes to which they were originally assigned. If these regions are exhausted, then the system restores contiguity by biasing the page replacement scheme to evict contiguous inactive pages. This system is implemented in FreeBSD on the Alpha architecture, and is evaluated on real applications and benchmarks. It is shown to yield substantial benefits when memory is plentiful and fragmentation is low. Furthermore, it sustains these benefits over the long term, by controlling the fragmentation arising from complex workload scenarios.

The contributions of this work are four-fold. It extends a previously proposed reservation-based approach to work with multiple, potentially very large superpage sizes, and demonstrates the benefits of doing so; it is, to our knowledge, the first to investigate the effect of fragmentation on superpages; it proposes a novel contiguity-aware page replacement algorithm to control fragmentation; and it tackles issues that have to date been overlooked but are required to make a solution practical, such as superpage demotion and eviction of dirty superpages.

Section 2 motivates the problem and establishes its constraints and complexities. Section 3 examines the related work on superpages. Section 4 and 5 describe our design and implementation, and Section 6 presents the results of an experimental evaluation. Finally, Section 7 concludes.

2 The superpage problem

This section discusses the motivation, hardware constraints, issues and tradeoffs in operating system support for superpages.

2.1 Motivation

Main memory has grown exponentially in size over at least the last decade and, as cause or consequence, the memory requirements of applications have proportionally increased [20]. In contrast, TLB coverage has lagged behind. The TLB is usually fully associative and its access time must be kept low, since it is in the critical path of every memory access [13]. Hence, TLB size has remained relatively small, usually 128 or fewer entries, corresponding to a megabyte or less of TLB coverage. Figure 1 depicts the TLB coverage achieved as a percentage of main memory size, for a number of Sun and SGI workstation models available between 1986 and 2001. Relative TLB coverage is seen to be decreasing by roughly a factor of 100 over ten years. As a consequence, many modern applications have working sets larger than the TLB coverage. Section 6.3 shows that for many real applications, TLB misses degrade performance by as much as 30% to 60%, contrasting to the 4% to 5% reported in the 1980's [2, 24] or the 5% to 10% reported in the 1990's [17, 23]. Another trend that has contributed to this performance degradation is that machines are now usually shipped with on-board, physically addressed caches that are larger than the TLB coverage. As a result, many TLB misses require access to the memory banks to find a translation for data that is already in the cache, making misses relatively more expensive.

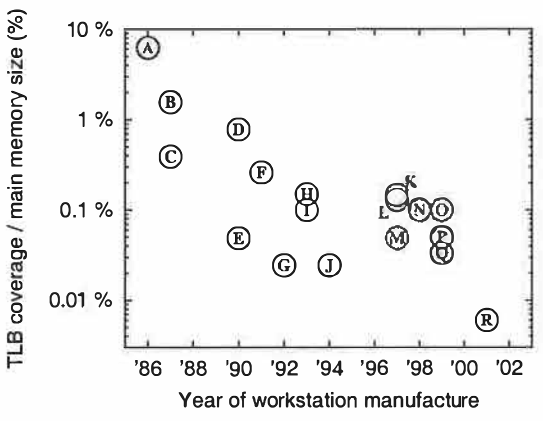


Figure 1: TLB coverage as percentage of main memory for workstations, 1986-2001 (data collected from various websites). (A) Sun 3/50; (B) Sun 3/180; (C) Sun 3/280; (D) Personal Iris; (E) SPARCstation-5; (F) Iris Indigo; (G) SPARCstation-10; (H) Indy; (I) Indigo2; (J) SPARCstation-20; (K) Ultra-1; (L) Ultra-2; (M) O2; (N) Ultra-5; (O) Ultra-10; (P) Ultra-60; (Q) Ultra-450; (R) Octane2.

We therefore seek a method of increasing TLB coverage without proportionally enlarging the TLB size. One option is to always use base pages of a larger size, say 64KB or 4MB. However, this approach would cause increased internal fragmentation due to partly used pages, and therefore induce premature onset of memory pressure [22]. Also, the I/O demands become higher due to increased paging granularity.

In contrast, the use of multiple page sizes enables an increase in TLB coverage while keeping internal fragmentation and disk traffic low. This technique, however, imposes several challenges upon the operating system designer, which are discussed in the rest of this section.

2.2 Hardware-imposed constraints

The design of TLB hardware in most processors imposes a series of constraints on superpages. Firstly, the superpage size must be among a set of page sizes supported by the processor. For example, the Alpha processor provides 8KB base pages and 64KB, 512KB and 4MB superpages; the i386 processor family supports 4KB and 4MB pages, and the new Itanium CPU provides ten different page sizes from 4KB to 256MB.

Secondly, a superpage is required to be contiguous in physical and virtual address space. Thirdly, its starting address in the physical and virtual address space must be a multiple of its size; for example, a 64KB superpage must be aligned on a 64KB address boundary.

Finally, the TLB entry for a superpage provides only a single reference bit, dirty bit, and set of protection attributes. The latter implies that all base pages that form a superpage must have the same protection attributes (read, write, execute). Also, due to the coarse granularity of reference and dirty bits, the operating system can determine whether some part of the superpage has been accessed or written to, but cannot distinguish between base pages in this regard.

2.3 Issues and tradeoffs

The task of managing superpages can be conceptually broken down into a series of steps, each governed by a different set of tradeoffs. The forthcoming analysis of these issues is independent of any particular processor architecture or operating system.

We assume that the virtual address space of each process consists of a set of virtual memory objects. A memory object occupies a contiguous region of the virtual address space and contains application-specific data, as shown in Figure 2. Examples of memory objects include memory mapped files, and the code, data, stack and heap segments of processes. Physical memory for these objects is allocated as and when their pages are first accessed.

Allocation: When a page in a memory object is first touched by the application, the OS allocates a physical page frame, and maps it into the application's address space. In principle, any available page frame can be used for this purpose, just as in a system without superpage support. However, should the OS later wish to create a superpage for the object, already allocated pages may require relocation (i.e., physical copying) to satisfy the contiguity and alignment constraints of superpages. The copying costs associated with this *relocation-based* allocation approach can be difficult to recover, especially on a busy system.

An alternative is *reservation-based* allocation. Here, the OS tries to allocate a page frame that is part of an available, contiguous range of page frames equal in size and alignment to the maximal desired superpage size, and tentatively reserves the entire set for use by the process. Subsequently, when the process first touches other pages that fall within the bounds of a reservation, the corresponding base page frames are allocated and mapped. Should the OS later decide to create a superpage for this object, the allocated page frames already satisfy the contiguity and alignment constraints. Figure 2 depicts this approach.

Reservation-based allocation requires the *a priori* choice of a superpage size to reserve, without foreknowledge of memory accesses to neighbouring pages. The OS may optimistically choose the desired superpage size as the largest supported size that is smaller or equal to the size of the memory object, but it may also bias this decision on the availability of contiguous physical memory. The OS must trade off the performance gains of using a large superpage against the option of retaining the contiguous region for later, possibly more critical use.

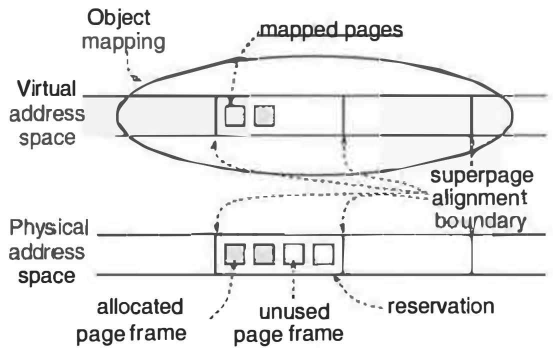


Figure 2: Reservation-based allocation.

Fragmentation control: When contiguous memory is plentiful, the OS succeeds in using superpages of the desired sizes, and achieves the maximum performance due to superpages. In practice, reservation-based allocation, use of different page sizes and file cache accesses have the combined effect of rapidly fragmenting available physical memory. To sustain the benefits of superpages, the OS may proactively release contiguous chunks of inactive memory from previous allocations, at the possible expense of having to perform disk I/O later. The OS may also preempt an existing, partially used reservation, given the possibility that the reservation may never become a superpage. The OS must therefore treat contiguity as a potentially contended resource, and trade off the impact of various contiguity restoration techniques against the benefits of using large superpages.

Promotion: Once a certain number of base pages within a potential superpage have been allocated, assuming that the set of pages satisfy the aforementioned constraints on size, contiguity, alignment and protection, the OS may decide to *promote* them into a superpage. This usually involves updating the page table entries for each of the constituent base pages of the superpage to reflect the new superpage size. Once the superpage has been created, a single TLB entry storing the translation for any address within the superpage suffices to map the entire superpage.

Promotion can also be performed incrementally. When a certain number of base pages have been allocated in a contiguous, aligned subset of a reservation, the OS may decide to promote the subset into a small superpage. These superpages may be progressively promoted to larger superpages, up to the size of the original reservation.

In choosing when to promote a partially allocated reservation, the OS must trade off the benefits of early promotion in terms of reduced TLB misses against the increased memory consumption that results if not all constituent pages of the superpage are used.

Demotion: *Superpage demotion* is the process of marking page table entries to reduce the size of a superpage, either to base pages or to smaller superpages. Demotion is appropriate when a process is no longer actively using all portions of a superpage, and memory pressure calls for the eviction of the unused base pages. One problem is that the hardware only maintains a single reference bit for the superpage, making it difficult for the OS to efficiently detect which portions of a superpage are actively used.

Eviction: Eviction of superpages is similar to the eviction of base pages. When memory pressure demands it, an inactive superpage may be evicted from physical memory, causing all of its constituent base page frames to become available. When an evicted page is later faulted in, memory is allocated and a superpage may be created in the same way as described earlier.

One complication arises when a dirty superpage is paged out. Since the hardware maintains only a single dirty bit, the superpage may have to be flushed out in its entirety, even though some of its constituent base pages may be clean.

Managing superpages thus involves a complex set of tradeoffs; other researchers have also alluded to some of these issues [12, 13]. The next section describes previous approaches to the problem, and Section 4 describes how our design effectively tackles all these issues.

3 Related approaches

Many operating systems use superpages for kernel segments and frame buffers. This section discusses existing superpage solutions for *application memory*, which is the focus of this work. These approaches can be classified by how they manage the contiguity required for superpages: reservation-based schemes try to preserve contiguity; relocation-based approaches create contiguity; and hardware-based mechanisms reduce or eliminate the contiguity requirement for superpages.

3.1 Reservations

Reservation-based schemes make superpage-aware allocation decisions at page-fault time. On each allocation, they use some policy to decide the preferred size of the allocation and attempt to find a contiguous region of free physical memory of that size.

Talluri and Hill propose a reservation-based scheme, in which a region is reserved at page-fault time and promoted when the number of frames in use reaches a promotion threshold. Under memory pressure, reservations can be preempted to regain free space [20]. The main goal of Talluri and Hill's design is to provide a simple, best-effort mechanism tailored to the use of partial-subblock TLBs, which are described in Section 3.3.

In contrast, superpages in both the HP-UX [19] and IRIX [6] operating systems are eagerly created at page-fault time. When a page is faulted in, the system may allocate several contiguous frames to fault in surrounding pages and immediately promote them into a superpage, regardless of whether the surrounding pages are likely to be accessed. Although pages are never actually reserved, this eager promotion mechanism is equivalent to a reservation-based approach with a promotion threshold of one frame.

In IRIX and HP-UX, the preferred superpage size is based on memory availability at allocation time, and on a user-specified per-segment page size hint. This hint is associated with an application binary's text and data segments; IRIX also allows the hint to be specified at runtime.

The main drawback of IRIX and HP-UX's eager promotion is that it is not transparent. It requires experimentation to determine the optimum superpage size for the various segments of a given application. A suboptimal setting will result in lower performance, due to either insufficient TLB coverage if superpages are too small, or unnecessary paging and page population costs if superpages are too large.

3.2 Page relocation

Relocation-based schemes create superpages by physically copying allocated page frames to contiguous regions when they determine that superpages are likely to be beneficial. Relocation-based approaches can be entirely and transparently implemented in the hardware-dependent layer of the operating system, but they need to relocate most of the allocated base pages of a superpage prior to promotion, even when there are plenty of contiguous available regions.

Romer et al. propose a competitive algorithm that uses online cost-benefit analysis to determine when the benefits of superpages outweigh the overhead of superpage promotion through relocation [16]. Their design requires a software-managed TLB, since it associates with each potential superpage a counter that must be updated by the TLB miss handler. In the absence of memory contention, this approach has a strictly lower performance than a reservation-based approach, because, in addition to the relocation costs, (1) there are more TLB misses, since relocation is performed as a reaction to an excessive number of TLB misses, and (2) TLB misses are more expensive — by a factor of four or more, according to Romer et al. — due to a more complex TLB miss handler. On the other hand, a relocation approach is more robust to fragmentation.

Reservations and page relocation can complement each other in a hybrid approach. One way would be to use relocation whenever reservations fail to provide enough contiguity and a large number of TLB misses is observed. Alternatively, page relocation can be performed as a background task to do *off-line memory compaction*. The goal is to merge fragmented chunks and gradually restore contiguity in the system. The IRIX *coalescing daemon* does this and is described in [6], but no evaluation is presented.

3.3 Hardware support

The contiguity requirement for superpages can be reduced or eliminated by means of additional hardware support.

Talluri and Hill study different TLB organizations. They advocate *partial-subblock* TLBs, which essentially contain superpage TLB entries that allow "holes" for missing base pages. They claim that with this approach most of the benefits from superpages can be obtained with minimal modifications to the operating system [20]. Partial-subblock TLBs yield only moderately larger TLB coverage than the base system, and it is not clear how to extend the partial-subblock TLBs to multiple superpage sizes.

Fang et al. describe a hardware-based mechanism that completely eliminates the contiguity requirement of superpages. They introduce an additional level of address translation in the memory controller, so that the operating system can promote non-adjacent physical pages into a superpage. This greatly simplifies the task of the operating system for supporting superpages [3].

To the best of our knowledge, neither partial-subblock TLBs nor address-remapping memory controllers are supported on commercial, general-purpose machines.

Our approach generalizes Talluri and Hill's reservation mechanism to multiple superpage sizes. To regain contiguity on fragmented physical memory without relocating pages, it biases the page replacement policy to select those pages that contribute the most to contiguity. It also tackles the issues of demotion and eviction (described in Section 2.3) not addressed by previous work, and does not require special hardware support.

4 Design

Our design adopts the reservation-based superpage management paradigm introduced in [20]. It extends the basic design along several dimensions, such as support for multiple superpage sizes, scalability to very large superpages, demotion of sparsely referenced superpages, effective preservation of contiguity without the need for compaction, and efficient disk I/O for partially modified superpages. As shown in Section 6, this combination of techniques is general enough to work efficiently for a range of realistic workloads, and is believed to be suitable for deployment in modern operating systems.

A high-level sketch of the design contains the following components. Available physical memory is classified into contiguous regions of different sizes, and is managed using a buddy allocator [14]. A multi-list reservation scheme is used to track partially used memory reservations, and to help in choosing reservations for preemption, as described in Section 4.8. A population map keeps track of memory allocations in each memory object, as described in Section 4.9. The system uses these data structures to implement allocation, preemption, promotion and demotion policies. Finally, it controls external memory fragmentation by performing page replacements in a contiguity-aware manner, as described in Section 4.4. The following subsections elaborate on these concepts.

4.1 Reservation-based allocation

Most operating systems allocate physical memory on application demand. When a virtual memory page is accessed by a program and no mapping exists in the page table, the OS's page fault handler is invoked. The handler attempts to locate the associated page in main memory; if it is not resident, an available page frame is allocated and the contents are either zero-filled or fetched from the paging device. Finally, the appropriate mapping is entered into the page table.

Instead of allocating physical memory one frame at a time, our system determines a preferred superpage size for the region encompassing the base page whose access caused the page fault. The choice of a size is made according to a policy described in Section 4.2. At page-fault time, the system obtains from the buddy allocator a set of contiguous page frames corresponding to the chosen superpage size. The frame with the same address alignment as the faulted page is used to fault in the page, and a mapping is entered into the page table for this page only. The entire set of frames is tentatively *reserved* for potential future use as a superpage, and added to a reservation list. In the event of a page fault on a page for which a frame has already been reserved, a mapping is entered into the page table for the base page.

4.2 Preferred superpage size policy

Next, we describe the policy used to choose the desired superpage size during allocation. Since this decision is usually made early in a process's execution, when it is hard to predict its future behaviour, our policy looks only at attributes of the memory object to which the faulting page belongs. If the chosen size turns out to be too large, then the decision will be later overridden by preempting the initial reservation. However, if the chosen size is too small, then the decision cannot be reverted without relocating pages. For that reason, the policy tends to choose the maximum superpage size that can be effectively used in an object.

For memory objects that are fixed in size, such as code segments and memory-mapped files, the desired reservation size is the largest, aligned superpage that contains the faulting page, does not overlap with existing reservations or allocated pages, and does not reach beyond the end of the object.

Dynamically sized memory objects such as stacks and heaps can grow one page at a time. Under the policy for fixed size objects, they would not be able to use superpages, because each time the policy would set the preferred size to one base page. Thus a slightly different policy is required. As before, the desired size is the largest, aligned superpage that contains the faulting page and does not overlap with existing reservations or allocations. However, the restriction that the reservation must not reach beyond the end of the object is dropped to allow for growth. To avoid wastage of contiguity for small objects that may never grow large, the size of this superpage is limited to the current size of the object. This policy thus uses large reservations only for objects that have already reached a sufficiently large size.

4.3 Preempting reservations

When free physical memory becomes scarce or excessively fragmented, the system can preempt frames that are reserved but not yet used. When an allocation is requested and no extent of frames with the desired size is available, the system has to choose between (1) refusing the allocation and thus reserving a smaller extent than desired, or (2) preempting an existing reservation that has enough unallocated frames to yield an extent of the desired size.

Our policy is that, whenever possible, the system preempts existing reservations rather than refusing an allocation of the desired size. When more than one reservation can yield an extent of the desired size, the reservation is preempted whose most recent page allocation occurred least recently, among all candidate reservations. This policy is based on the observation that useful reservations are often populated quickly, and that reservations that have not experienced any recent allocations are less likely to be fully allocated in the near future.

4.4 Fragmentation control

Allocating physical memory in contiguous extents of multiple sizes leads to fragmentation of main memory. Over time, extents of large sizes may become increasingly scarce, thus preventing the effective use of superpages.

To control fragmentation, our buddy allocator performs coalescing of available memory regions whenever possible. However, coalescing by itself is only effective if the system periodically reaches a state where all or most of main memory is available. To control fragmentation under persistent memory pressure, the page replacement daemon is modified to perform contiguity-aware page replacement. Section 5.1 discusses this in greater detail.

4.5 Incremental promotions

A superpage is created as soon as any superpage-sized and aligned extent within a reservation gets fully populated. Promotion, therefore, is incremental: if, for instance, pages of a memory object are faulted in sequentially, a promotion occurs to the smallest superpage size as soon as the population count corresponds to that size. Then, when the population count reaches the next larger superpage size, another promotion occurs to the next size, and so on.

It is possible to promote to the next size when the population count reaches a certain fraction of that size. However, before performing the promotion the system needs to populate the entire region, which could artificially inflate the memory footprint of applications. We promote only regions that are fully populated by the application, since we observe that most applications populate their address space densely and relatively early in their execution.

4.6 Speculative demotions

Demotion occurs as a side-effect of page replacement. When the page daemon selects a base page for eviction that is part of a superpage, the eviction causes a demotion of that superpage. This demotion is also incremental, since it is not necessary to demote a large superpage all the way to base pages just because one of its constituent base pages is evicted. Instead, the superpage is first demoted to the next smaller superpage size, then the process is applied recursively for the smaller superpage that encompasses the victim page, and so on. Demotion is also necessary whenever the protection attributes are changed on part of a superpage. This is required because the hardware provides only a single set of protection bits for each superpage.

The system may also periodically demote active superpages *speculatively* in order to determine if the superpage is still being actively used in its entirety. Recall that the hardware only provides a single reference bit with each superpage. Therefore, the operating system has no way to distinguish a superpage in which all the constituent base pages are being accessed, from one in which only a subset of the base pages are. In the latter case, it would be desirable to demote the superpage under memory pressure, such that the unused base pages can be discovered and evicted.

To address this problem, when the page daemon resets the reference bit of a superpage's base page, and if there is memory pressure, then it recursively demotes the superpage that contains the chosen base page, with a certain probability p. In our current implementation, p is 1. Incremental repromotions occur when all the base pages of a demoted superpages are being referenced.

4.7 Paging out dirty superpages

When a dirty superpage needs to be written to disk, the operating system does not possess dirty bit information for individual base pages. It must therefore consider all the constituent base pages dirty, and write out the superpage in its entirety, even though only a few of its base pages may have actually been modified. For large, partially dirty superpages, the performance degradation due to this superfluous I/O can considerably exceed any benefits from superpages.

To prevent this problem, we demote clean superpages whenever a process attempts to write into them, and repromote later if all the base pages are dirtied. This choice is evaluated in Section 6.7.

Inferring dirty base pages using hash digests: As an alternative, we considered a technique that retains the benefits of superpages even when they are partially dirty, while avoiding superfluous I/O. When a clean memory page is read from disk, a cryptographic hash digest of its contents is computed and recorded. If a partially dirty set of base pages is promoted to a superpage, or if a clean superpage becomes dirty, then all its constituent base pages are considered dirty. However, when the page is flushed out, the hash of each base page is recomputed and compared to determine if it was actually modified and must be written to disk.

A 160-bit SHA-1 hash has a collision probability of about one in 2^{80} [4], which is much smaller than the probability of a hardware failure. Hence this technique can be considered safe. However, preliminary microbenchmarks using SHA-1 reveal significant overhead, up to 15 %, on disk-intensive applications. The pathological case of a large sequential read when the CPU is saturated incurs a worst-case degradation of 60%. Therefore, we did not use this technique in our implementation.

However, these overheads can be reduced using a variety of optimizations. First, the hash computation can be postponed until there is a partially dirty superpage, so that fully-clean or fully-dirty superpages and unpromoted base pages need not be hashed. Second, the hashing cost can be eliminated from the critical path by performing it entirely from the idle loop, since the CPU may frequently be idle for disk-intensive workloads. An evaluation of these optimizations is the subject of future work.

4.8 Multi-list reservation scheme

Reservation lists keep track of reserved page frame extents that are not fully populated. There is one reservation list for each page size supported by the hardware, except for the largest superpage size. Each reservation appears in the list corresponding to the size of the largest free extent that can be obtained if the reservation is preempted. Because a reservation has at least one of its frames allocated, the largest extents it can yield if preempted are one page size smaller than its own size. For instance, on an implementation for the Alpha processor, which supports 4MB, 512KB, 64KB and 8KB pages, the 64KB reservation list may contain reservations of size 512KB and 4MB.

Reservations in each list are kept sorted by the time of their most recent page frame allocations. When the system decides to preempt a reservation of a given size, it chooses the reservation at the head of the list for that size. This satisfies our policy of preempting the extent whose most recent allocation occurred least recently among all reservations in that list.

Preempting a chosen reservation occurs as follows. Rather than breaking the reservation into base pages, it is broken to smaller extents. Unpopulated extents are transferred to the buddy allocator and partially populated ones are reinserted into the appropriate lists. For example, when preempting a 512KB reservation taken from head of the 64KB list, the reservation is broken into eight 64KB extents. The ones with no allocations are freed and the ones that are partially populated are inserted at the head of the 8KB reservation list. Fully populated extents are not reinserted into the reservation lists.

When the system needs a contiguous region of free memory, it can obtain it from the buddy allocator or by preempting a reservation. The mechanism is best described with an example. Still in the context of the Alpha CPU, suppose that an application faults in a given page for which there is no reserved frame. Further assume that the preferred superpage size for the faulting page is 64KB. Then the system first asks the buddy allocator for a 64KB extent. If that fails, it preempts the first reservation in the 64KB reservation list, which should yield at least one 64KB extent. If the 64KB list is empty, the system will try the 512KB list. If that list is also empty, then the system has to resort to base pages: the buddy allocator is tried first, and then the 8KB reservation list as the last resource.

4.9 Population map

Population maps keep track of allocated base pages within each memory object. They serve four distinct purposes: (1) on each page fault, they enable the OS to map the virtual address to a page frame that may already be reserved for this address; (2) while allocating contiguous regions in physical address space, they enable the OS to detect and avoid overlapping regions; (3) they assist in making page promotion decisions; and (4) while preempting a reservation, they help in identifying unallocated regions.

A population map needs to support efficient lookups, since it is queried on every page fault. We use a radix tree in which each level corresponds to a page size. The root corresponds to the maximum superpage size supported by the hardware, each subsequent level corresponds to the next smaller superpage size, and the leaves correspond to the base pages. If the virtual pages represented by a node have a reserved extent of frames, then the node has a pointer to the reservation and the reservation has a back pointer to the node.

Each non-leaf node keeps a count of the number of superpage-sized virtual regions at the next lower level that have a population of at least one (the `somepop` counter), and that are fully populated (the `fullpop` counter), respectively. This count ranges from 0 through R, where R is the ratio between consecutive superpage sizes (8 on the Alpha processor). The tree is lazily updated as the object's pages are populated. The absence of a child node is equivalent to having a child with both counters zero. Since counters refer to superpage-sized regions, upward propagation of the counters occurs only when `somepop` transitions between 0 and 1, or when `fullpop` transitions between $R-1$ and R. Figure 3 shows one such tree.

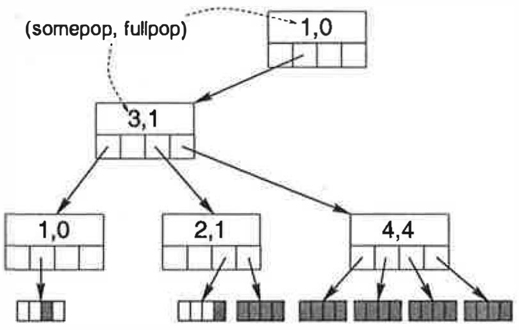


Figure 3: A population map. At the base page level, the actual allocation of pages is shown.

A hash table is used to locate population maps. For each population map, there is an entry associating a *memory_object, page_index* tuple with the map, where *page_index* is the offset of the starting page of the map within the object. The population map is used as follows:

Reserved frame lookup: On a page fault, the virtual address of the faulting page is rounded down to a multiple of the largest page size, converted to the corresponding *memory_object, page_index* tuple, and hashed to determine the root of the population map. From the root, the tree is traversed to locate the reserved page frame, if there is one.

Overlap avoidance: If the above procedure yields no reserved frame, then we attempt to make a reservation. The maximum size that does not overlap with previous reservations or allocations is given by the first node in the path from the root whose `somepop` counter is zero.

Promotion decisions: After a page fault is serviced, a promotion is attempted at the first node on the path from the root to the faulting page that is fully populated and has an associated reservation. The promotion attempt succeeds only if the faulting process has the pages mapped with uniform protection attributes and dirty bits.

Preemption assistance: When a reservation is preempted it is broken into smaller chunks that need to be freed or reinserted in the reservation lists, depending on their allocation status, as described in Section 4.8. The allocation status corresponds to the population counts in the superpage map node to which the reservation refers.

5 Implementation notes

This section describes some implementation specific issues of our design. While the discussion of our solution is necessarily OS-specific, the issues are general.

5.1 Contiguity-aware page daemon

FreeBSD's page daemon keeps three lists of pages, each in approximate LRU (A-LRU) order: active, inactive and cache. Pages in the cache list are clean and unmapped and hence can be easily freed under memory pressure. Inactive pages are those mapped into the address space of some process, and have not been referenced for a long time. Active pages are those that have been accessed recently, but may or may not have their reference bit set. Under memory pressure, the daemon moves clean inactive pages to the cache, pages out dirty inactive pages, and also deactivates some unreferenced pages from the active list. We made the following changes to factor contiguity restoration into the page replacement policy.

(1) We consider cache pages as available for reservations. The buddy allocator keeps them coalesced with the free pages, increasing the available contiguity of the system. These coalesced regions are placed at the tail of their respective lists, so that subsequent allocations tend to respect the A-LRU order.

The contents of a cache page are retained as long as possible, whether it is in a buddy list or in a reservation. If a cache page is referenced, then it is removed from the buddy list or the reservation; in the latter case, the reservation is preempted. The cache page is reactivated and its contents are reused.

(2) The page daemon is activated not only on memory pressure, but also when available contiguity falls low. In our implementation, the criterion for low contiguity is the failure to allocate a contiguous region of the preferred size. The goal of the daemon is to restore the contiguity that would have been necessary to service the requests that failed since the last time the daemon was woken. The daemon then traverses the inactive list and moves to the cache only those pages that contribute to this goal. If it reaches the end of the list before fulfilling its goal, then it goes to sleep again.

(3) Since the chances of restoring contiguity are higher if there are more inactive pages to choose from, all clean pages backed by a file are moved to the inactive list as soon as the file is closed by all processes. This differs from the current behaviour of FreeBSD, where a page does not change its status on file closing or process termination, and active pages from closed files may never be deactivated if there is no memory pressure. In terms of overall performance, our system thus finds it worthwhile to favor the likelihood of recovering the contiguity from these file-backed pages, than to keep them for a longer time for the chance that the file is accessed again.

Controlling fragmentation comes at a price. The more aggressively the system recovers contiguity, the greater is the possibility and the extent of a performance penalty induced by the modified page daemon, due to its deviation from A-LRU. Our modified page daemon aims at balancing this tradeoff. Moreover, by judiciously selecting pages for replacement, it attempts to restore as much contiguity as possible by affecting as few pages as possible. Section 6.5 demonstrates the benefits of this design.

5.2 Wired page clustering

Memory pages that are used by FreeBSD for its internal data structures are *wired*, that is, marked as non-pageable since they cannot be evicted. At system boot time these pages are clustered together in physical memory, but as the kernel allocates memory while other processes are running, they tend to get scattered. Our system with 512MB of main memory is found to rapidly reach a point where most 4MB chunks of physical memory contain at least one wired page. At this point, contiguity for large pages becomes irrecoverable.

To avoid this fragmentation problem, we identify pages that are about to be wired for the kernel's internal use. We cluster them in pools of contiguous physical memory, so that they do not fragment memory any more than necessary.

5.3 Multiple mappings

Two processes can map a file into different virtual addresses. If the addresses differ by, say, one base page, then it is impossible to build superpages for that file in the page tables of both processes. At most one of the processes can have alignment that matches the physical address of the pages constituting the file; only this process is capable of using superpages.

Our solution to this problem leverages the fact that applications most often do not specify an address when mapping a file. This gives the kernel the flexibility to assign a virtual address for the mapping in each process. Our system then chooses addresses that are compatible with superpage allocation. When mapping a file, the system uses a virtual address that aligns to the largest superpage that is smaller than the size of the mapping, thus retaining the ability to create superpages in each process.

6 Evaluation

This section reports results of experiments that exercise the system on several classes of benchmarks and real applications. We evaluate the best-case benefits of superpages in situations when system memory is plentiful. Then, we demonstrate the effectiveness of our design, by showing how these benefits are sustained despite different kinds of stress on the system. Results show the efficiency of our design by measuring its overhead in several pathological cases, and justify the design choices in the previous section using appropriate measurements.

6.1 Platform

We implemented our design in the FreeBSD-4.3 kernel as a loadable module, along with hooks in the operating system to call module functions at specific points. These points are page faults, page allocation and deallocation, the page daemon, and at the physical layer of the VM system (to demote when changing protections and to keep track of dirty/modified bits of superpages). We were also able to seamlessly integrate this module into the kernel. The implementation comprises of around 3500 lines of C code.

We used a Compaq XP-1000 machine with the following characteristics:

- Alpha 21264 processor at 500 MHz;
- four page sizes: 8KB base pages, 64KB, 512KB and 4MB superpages;
- fully associative TLB with 128 entries for data and 128 for instructions;
- software page tables, with firmware-based TLB loader;
- 512MB RAM;
- 64KB data and 64KB instruction L1 caches, virtually indexed and 2-way associative;
- 4MB unified, direct-mapped external L2 cache.

The Alpha firmware implements superpages by means of *page table entry (PTE) replication*. The page table stores an entry for every base page, whether or not it is part of a superpage. Each PTE contains the translation information for a base page, along with a page size field. In this PTE replication scheme, the promotion of a 4MB region involves the setting of the page size field of *each of the 512 page table entries* that map the region [18].

6.2 Workloads

We used the following benchmarks and applications to evaluate our system.

CINT2000: SPEC CPU2000 integer benchmark suite [7].

CFP2000: SPEC CPU2000 floating-point benchmark suite [7].

Web: The thttpd web server [15] servicing 50000 requests selected from an access log of the CS departmental web server at Rice University. The working set size of this trace is 238MB, while its data set is 3.6GB.

Image: 90-degree rotation of a 800x600-pixel image using the popular open-source ImageMagick tools [8].

Povray: Ray tracing of a simple image.

Linker: Link of the FreeBSD kernel with the GNU linker.

C4: An alpha-beta search solver for a 12-ply position of the connect-4 game, also known as the fhourstones benchmark.

Tree: A synthetic benchmark that captures the behaviour of processes that use dynamic allocation for a large number of small objects, leading to poor locality of reference. The benchmark consists of four operations performed randomly on a 50000-node red-black tree: 50% of the operations are lookups, 24% insertions, 24% deletions, and 2% traversals. Nodes on the tree contain a pointer to a 128-byte record. On insertions a new record is allocated and initialized; on lookups and traversals, half of the record is read.

SP: The sequential version of a scalar pentadiagonal uncoupled equation system solver, from the NAS Parallel Benchmark suite [1]. The input size corresponds to the "workstation class" in NAS's nomenclature.

FFTW: The Fastest Fourier Transform in the West [5] with a 200x200x200 matrix as input.

Matrix: A non-blocked matrix transposition of a 1000x1000 matrix.

6.3 Best-case benefits due to superpages

This first set of experiments shows that several classes of real workloads yield large benefits with superpages when free memory is plentiful and non-fragmented. Table 1 presents these best-case speedups obtained when the benchmarks are given the contiguous memory regions they need, so that every attempt to allocate regions of the preferred superpage size (as defined in Section 4.2) succeeds, and reservations are never preempted.

The speedups are computed against the unmodified system using the mean elapsed runtime of three runs after an initial warm-up run. For both the CINT2000 and CFP2000 entries in the table, the speedups reflect, respectively, the improvement in SPECint2000 and SPECfp2000 (defined by SPEC as the geometric mean of the normalized throughput ratios).

The table also presents the superpage requirements of each of these applications (as a snapshot measured at peak memory usage), and the percentage data TLB miss reduction achieved with superpages. In most cases the data TLB misses are virtually eliminated by superpages, as indicated by a miss reduction close to 100%. The contribution of instruction TLB misses to the total number of misses was found to be negligible in all of the benchmarks.

Bench-mark	Superpage usage				Miss reduc (%)	Speed-up
	8 KB	64 KB	512 KB	4 MB		
CINT2000						**1.112**
gzip	204	22	21	42	80.00	1.007
vpr	253	29	27	9	99.96	1.383
gcc	1209	1	17	35	70.79	1.013
mcf	206	7	10	46	99.97	1.676
crafty	147	13	2	0	99.33	1.036
parser	168	5	14	8	99.92	1.078
eon	297	6	0	0	0.00	1.000
perl	340	9	17	34	96.53	1.019
gap	267	8	7	47	99.49	1.017
vortex	280	4	15	17	99.75	1.112
bzip2	196	21	30	42	99.90	1.140
twolf	238	13	7	0	99.87	1.032
CFP2000						**1.110**
wupw	219	14	6	43	96.77	1.009
swim	226	16	11	46	98.97	1.034
mgrid	282	15	5	13	98.39	1.000
applu	1927	1647	90	5	93.53	1.020
mesa	246	13	8	1	99.14	0.985
galgel	957	172	68	2	99.80	1.289
art	163	4	7	0	99.55	1.122
equake	236	2	19	9	97.56	1.015
facerec	376	8	13	2	98.65	1.062
ammp	237	7	21	7	98.53	1.080
lucas	314	4	36	31	99.90	1.280
fma3d	500	17	27	22	96.77	1.000
sixtr	793	81	29	1	87.50	1.043
apsi	333	5	5	47	99.98	1.827
Web	30623	5	143	1	16.67	1.019
Image	163	1	17	7	75.00	1.228
Povray	136	6	17	14	97.44	1.042
Linker	6317	12	29	7	85.71	1.326
C4	76	2	9	0	95.65	1.360
Tree	207	6	14	1	97.14	1.503
SP	151	103	15	0	99.55	1.193
FFTW	160	5	7	60	99.59	1.549
Matrix	198	12	5	3	99.47	7.546

Table 1: Speedups and superpage requirements when plenty of memory is available.

Nearly all the workloads in the table display benefits due to superpages; some of these are substantial. Out of our 35 benchmarks, 18 show improvements over 5% (speedup of 1.05), and 10 show over 25%. The only application that slows down is mesa, which degrades by a negligible fraction. Matrix, with a speedup of 7.5, is close to the maximum potential benefits that can possibly be gained with superpages, because of its access pattern that produces one TLB miss for every two memory accesses.

Several commonplace desktop applications like Linker (gnuld), gcc, and bzip2 observe significant performance improvements. If sufficient contiguous memory is available, then these applications stand to benefit from a superpage management system. In contrast, Web gains little, because the system cannot create enough superpages in spite of its large 315MB footprint. This is because Web accesses a large number of small files, and the system does not attempt to build superpages that span multiple memory objects. Extrapolating from the results, a system without such limitation (which is technically feasible, but likely at a high cost in complexity) would bring Web's speedup closer to a more attractive 15%, if it achieved a miss reduction close to 100%.

Some applications create a significant number of large superpages. FFTW, in particular, stands out with 60 superpages of size 4MB. The next section shows that FFTW makes good use of large superpages, as there is almost no speedup if 4MB pages are not supported.

Mesa shows a small performance degradation of 1.5%. This was determined to be not due to the overhead of our implementation, but because our allocator does not differentiate zeroed-out pages from other free pages. When the OS allocates a page that needs to be subsequently zeroed out, it requests the memory allocator to preferentially allocate an already zeroed-out page if possible. Our implementation of the buddy allocator ignores this hint; we estimated the cost of this omission by comparing base system performance with and without the zeroed-page feature. We obtained an average penalty of 0.9%, and a maximum of 1.7%.

A side effect of using superpages is that it subsumes page coloring [11], a technique that FreeBSD and other operating systems use to reduce cache conflicts in physically-addressed and especially in direct-mapped caches. By carefully selecting among free frames when mapping a page, the OS keeps virtual-to-physical mappings in a way such that pages that are consecutive in virtual space map to consecutive locations in the cache. Since with superpages virtually contiguous pages map to physically contiguous frames, they automatically map to consecutive locations in a physically-mapped cache. Our speedup results factor out the effect of page-coloring, because the benchmarks were run with enough free memory for the unmodified system to always succeed in its page coloring attempts. Thus, both the unmodified and the modified system effectively benefit from page coloring.

6.4 Benefits from multiple superpage sizes

We repeated the above experiments, but changed the system to support only one superpage size, for each of 64KB, 512KB and 4MB, and compared the resulting performance against our multi-size implementation. Tables 2 and 3 respectively present the speedup and TLB miss reduction for the benchmarks, excluding those that have the same speedup (within 5%) in all four cases.

Benchmark	64KB	512KB	4MB	All
CINT2000	1.05	1.09	1.05	1.11
vpr	1.28	1.38	1.13	1.38
mcf	1.24	1.31	1.22	1.68
vortex	1.01	1.07	1.08	1.11
bzip2	1.14	1.12	1.08	1.14
CFP2000	1.02	1.08	1.06	1.12
galgel	1.28	1.28	1.01	1.29
lucas	1.04	1.28	1.24	1.28
apsi	1.04	1.79	1.83	1.83
Image	1.19	1.19	1.16	1.23
Linker	1.16	1.26	1.19	1.32
C4	1.30	1.34	0.98	1.36
SP	1.19	1.17	0.98	1.19
FFTW	1.01	1.00	1.55	1.55
Matrix	3.83	7.17	6.86	7.54

Table 2: Speedups with different superpage sizes.

The results show that the best superpage size depends on the application. For instance, it is 64KB for SP, 512KB for vpr, and 4MB for FFTW. The reason is that while some applications only benefit from large superpages, others are too small to fully populate large superpages. To use large superpages with small applications, the population threshold for promotion could be lowered, as suggested in Section 4.5. However, the OS would have to populate regions that are only partially mapped by the application. This would enlarge the application footprint, and also slightly change the OS semantics, since some invalid accesses would not be caught.

The tables also demonstrate that allowing the system to choose between multiple page sizes yields higher performance, because the system dynamically selects the best size for every region of memory. An extreme case is mcf, for which the percentage speedup when the system gets to choose among several sizes more than doubles the speedup with any single size.

Some apparent anomalies, like different speedups with the same TLB miss reduction (e.g., Linker) are likely due to the coarse granularity of the Alpha processor's TLB miss counter (512K misses). For short-running benchmarks, 512K misses corresponds to a two-digit percentage of the total number of misses.

Benchmark	64KB	512KB	4MB	All
CINT2000				
vpr	82.49	98.66	45.16	99.96
mcf	55.21	84.18	53.22	99.97
vortex	46.38	92.76	80.86	99.75
bzip2	99.80	99.09	49.54	99.90
CFP2000				
galgel	98.51	98.71	0.00	99.80
lucas	12.79	96.98	87.61	99.90
apsi	9.69	98.70	99.98	99.98
Image	50.00	50.00	50.00	75.00
Linker	57.14	85.71	57.14	85.71
C4	95.65	95.65	0.00	95.65
SP	99.11	93.75	0.00	99.55
FFTW	7.41	7.41	99.59	99.59
Matrix	90.43	99.47	99.47	99.47

Table 3: TLB miss reduction percentage with different superpage sizes.

6.5 Sustained benefits in the long term

The performance benefits of superpages can be substantial, provided contiguous regions of physical memory are available. However, conventional systems can be subject to memory fragmentation even under moderately complex workloads. For example, we ran instances of grep, emacs, netscape and a kernel compilation on a freshly booted system; within about 15 minutes, we observed severe fragmentation. The system had completely exhausted all contiguous memory regions larger than 64KB that were candidates for larger superpages, even though as much as 360MB of the 512MB were free.

Our system seeks to preserve the performance of superpages over time, so it actively restores contiguity using techniques described in Sections 4.4 and 5.1. To evaluate these methods, we first fragment the system memory by running a web server and feeding it with requests from the same access log as before. The file-backed memory pages accessed by the web server persist in memory and reduce available contiguity to a minimum. Moreover, the access pattern of the web server results in an interleaved distribution of active, inactive and cache pages, which increases fragmentation.

We present two experiments using this web server.

Sequential execution: After the requests from the trace have been serviced, we run the FFTW benchmark four times in sequence. The goal is to see how quickly the system recovers just enough contiguous memory to build superpages and perform efficiently.

Figure 4 compares the performance of two contiguity restoration techniques. The *cache* scheme treats all cached pages as available, and coalesces them into the buddy allocator. The graph depicts no appreciable performance improvements of FFTW over the base system. We observed that the system is unable to provide even a single 4MB superpage for FFTW. This is because memory is available (47MB in the first run and 290MB in the others), but is fragmented due to active, inactive and wired pages.

The other scheme, called *daemon*, is our implementation of contiguity-aware page replacement and wired page clustering. The first time FFTW runs after the web server, the page daemon is activated due to contiguity shortage, and is able to recover 20 out of the requested 60 contiguous regions of 4MB size. Subsequent runs get a progressively larger number of 4MB superpages, viz. 35, 38 and 40. Thus, FFTW performance reaches near-optimum within two runs, i.e., a speedup of 55%.

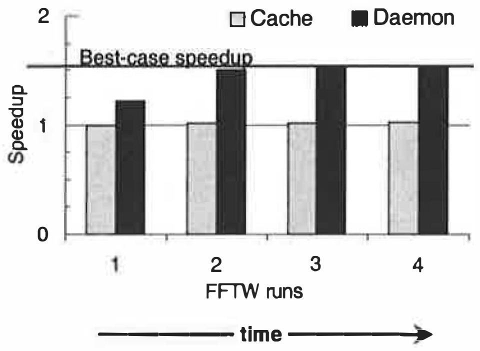


Figure 4: Two techniques for fragmentation control.

The web server closes its files on exit, and our page daemon treats this file memory as inactive, as described in Section 5.1. We now measure the impact of this effect in conjunction with the page daemon's drive to restore contiguity, on the web server's subsequent performance. We run the web server again after FFTW, and replay the same trace. We observe only a 1.6% performance degradation over the base system, indicating that the penalty on the web server performance is small.

We further analyze this experiment by monitoring the available contiguity in the system over time. We define an empirical *contiguity metric* as follows. We assign 1, 2 or 3 points to each base page that belongs to a 64KB, 512KB, or 4MB memory region respectively, assuming that the region is contiguous, aligned and fully available. We compute the sum of these per-page points, and normalize it to the corresponding value if every page in the system were to be free. Figure 5 shows a plot of this contiguity metric against experimental time. Note that this metric is unfavorable to the daemon scheme since it does not consider as available the extra contiguity that can be regained by moving inactive pages to the cache.

At the start of the experiment, neither scheme has all of the system's 512MB available; in particular, the cache scheme has lost 5% more contiguity due to unclustered wired pages. For about five minutes, the web server consumes memory and decreases available contiguity to zero. Thereafter, the cache scheme recovers only 8.8% of the system's contiguity, which can be seen in the graph as short, transitory bursts between FFTW executions. In contrast, the daemon scheme recovers as much as 42.4% of the contiguity, which is consumed by FFTW while it executes, and released each time it exits. The FFTW executions thus finish earlier, at 8.5 minutes for the daemon scheme, compared to 9.8 minutes for the cache scheme.

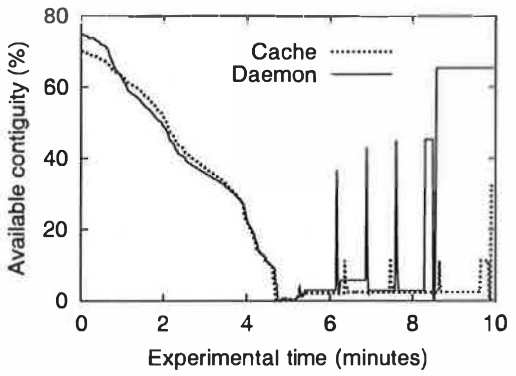


Figure 5: Contiguity as a function of time.

To estimate the maximum contiguity that can be potentially gained back after the FFTW runs complete, we run a synthetic application that uses enough anonymous memory to maximize the number of free pages in the system when it exits. At this point, the amount of contiguity lost is 54% in the cache scheme, mostly due to scattered wired pages. In contrast, the daemon scheme in unable to recover 13% of the original contiguity. The reason is that the few active and inactive pages that remain at the end of the experiment are scattered in physical memory over as many as 54 4MB chunks. Since the experiment starts on a freshly booted system, active and inactive pages were physically close at that time, occupying only 22 such chunks. Part of the lost 13% is due to inactive pages that are not counted in the contiguity metric, but can be recovered by the page daemon. Therefore, the real loss in the long term for the daemon scheme is bounded only by the number of active pages.

Concurrent execution: The next experiment runs the web server *concurrently* with a contiguity-seeking application. The goal is to measure the effect of the page replacement policy on the web server during a single, continuous run. We isolate the effect of the page replacement policy by disabling superpage promotions in this experiment.

We warm up the web server footprint by playing 100,000 requests from the trace, and then measure the time taken to service the next 100,000 requests. We wish to avoid interference of the CPU-intensive FFTW application with the web server, so we substitute it with a dummy application that only exercises the need for contiguity. This application maps, touches and unmaps 1MB of memory, five times a second, and forces the page daemon to recover contiguity rather than just memory.

The web server keeps its active files open while it is running, so our page daemon cannot indiscriminately treat this memory as inactive. The web server's active memory pages get scattered, and only a limited amount of contiguity can be restored without compacting memory. Over the course of the experiment, the dummy application needs about 3000 contiguous chunks of 512KB size. The original page daemon only satisfied 3.3% of these requests, whereas our contiguity-aware page daemon fulfills 29.9% of the requests. This shows how the change in the replacement policy succeeds in restoring significantly more contiguity than before, with negligible overhead and essentially no performance penalty.

The overhead of the contiguity restoration operations of the page daemon is found to be only 0.8%, and the web server suffers an additional 3% of performance degradation, as a consequence of the deviation of the page replacement policy from A-LRU.

6.6 Adversary applications

This section exercises the system on three synthetic pathological workloads, and concludes with a measurement of realistic overhead.

Incremental promotion overhead: We synthesized an adversary application that makes the system pay all the costs of incremental promotion without gaining any benefit. It allocates memory, accesses one byte in each page, and deallocates the memory, which renders the TLB useless since every translation is used only once. This adversary shows a slowdown of 8.9% with our implementation, but as much as 7.2% of this overhead is due to the following hardware-specific reason. PTE replication, as described in Section 6.1, forces each page table entry to be traversed six times: once per each of the three incremental promotions, and once per each of the three incremental demotions. The remaining 1.7% of the overhead is mainly due to maintenance of the population maps.

Sequential access overhead: Accessing pages sequentially as in our adversary is actually a common behaviour, but usually every byte of each page is accessed, which dilutes the overhead. We tested the `cmp` utility, which compares two files by mapping them in memory, using two identical 100MB files as input, and observed a negligible performance degradation of less than 0.1%.

Preemption overhead: To measure the overhead of preempting reservations, we set up a situation where there is only 4MB of memory available and contiguous, and run a process that touches memory with a 4MB stride. In this situation, there is a pattern of one reservation preemption every seven allocations. Every preemption splits a reservation into 8 smaller chunks. One remains reserved with the page that made the original reservation; another is taken for the page being allocated, and 6 are returned to the free list. We measured a performance degradation of 1.1% for this process.

Overhead in practice: Finally, we measure the total overhead of our implementation in real scenarios. We use the same benchmarks of Section 6.2, perform all the contiguous memory allocation and fragmentation management as before, but factor out the benefit of superpages by simply not promoting them. We preserve the promotion overhead by writing the new superpage size into some unused portion of the page table entries. We observe performance degradations of up to 2%, with an average of about 1%. This shows how our system only imposes negligible overhead in practice, so the pathological situations described above are rarely observed.

6.7 Dirty superpages

To evaluate our decision of demoting clean superpages upon writing, as discussed in Section 4.7, we coded a program that maps a 100MB file, reads every page thus triggering superpage promotion, then writes into every 512th page, flushes the file and exits. We compared the running time of the process both with and without demoting on writing. As expected, since the I/O volume is 512 times larger, the performance penalty of not demoting is huge: a factor of more than 20.

Our design decision may deny the benefits of superpages to processes that do not write to all of the base pages of a potential superpage. However, according to our policy, we choose to pay that price in order to keep the degradation in pathological cases low.

6.8 Scalability

If the historical tendencies of decreasing relative TLB coverage and increasing working set sizes continue, then to keep TLB miss overhead low, support for superpages much larger than 4 MB will be needed in the future. Some processors like the Itanium and the Sparc64-III provide 128MB and larger superpages, and our superpage system is designed to scale to such sizes. However, architectural peculiarities may pose some obstacles.

Most operations in our implementation are either $O(1)$; or $O(S)$, where S is the number of distinct superpage sizes; or in the case of preempting a reservation, $O(S{*}R)$, where R is the ratio between consecutive sizes, which is never more than 8 on modern processors. The exceptions are four routines with running time linear in the size (in base pages) of the superpage that they operate on. One is the page daemon that scans pages; since it runs as a background process, it is not in the critical path of memory accesses. The other three routines are promotion, demotion, and dirty/reference bit emulation. They operate on each page table entry in the superpage, and owe their unscalability to the hardware-defined PTE replication scheme described in Section 6.1.

Promotions and demotions: Often, under no memory pressure, pages are incrementally promoted early in a process's life and only demoted at program exit. In such case, the cost amortized over all pages used by the process is $O(S)$, which is negligible in all of our benchmarks. The only exception to this is the adversary experiment of Section 6.6, which pays a 7.2% overhead due to incremental promotions and demotions. However, when there is memory pressure, demotions and repromotions may happen several times in a process's life (as described in Sections 4.6 and 4.7). The cost of such operations may become significant for very large superpages, given the linear cost of PTE replication.

Dirty/reference bit emulation: In many processors, including the Alpha, dirty and reference bits must be emulated by the operating system. This emulation is done by protecting the page so that the first write or reference triggers a software trap. The trap handler registers in the OS structures that the page is dirty or referenced, and resets the page protection. For large superpages, setting and resetting protection can be expensive if PTE replication is required, as it must be done for every base page.

These problems motivate the need for more superpage-friendly page table structures, whether they are defined by the hardware or the OS, in order to scalably support very large superpages. *Clustered page tables* proposed by Talluri et al. [21] represent one step in this direction.

7 Conclusions

This paper provides a transparent and effective solution to the problem of superpage management in operating systems. Superpages are physical pages of large size, which may be used to increase TLB coverage, reduce TLB misses, and thus improve application performance. We describe a practical design and demonstrate that it can be integrated into an existing general-purpose operating system. We evaluate the system on a range of real workloads and benchmarks, observe performance benefits of 30% to 60% in several cases, and show that the system is robust even in pathological cases. These benefits are sustained under complex workload conditions and memory pressure, and the overheads are small.

Acknowledgments

We wish to thank our shepherd Greg Ganger and the anonymous referees for their helpful comments. This work was supported in part by NSF grant CCR-98110603, Texas ATP grant 003604-0150-1999, and equipment donations from Compaq WRL and HP Labs. Juan Navarro was supported in part by a USENIX Student Research Grant.

References

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995.

[2] D. W. Clark and J. S. Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *ACM Transactions Computer Systems*, 3(1):31–62, Feb. 1985.

[3] Z. Fang, L. Zhang, J. Carter, S. McKee, and W. Hsieh. Reevaluating online superpage promotion with hardware support. In *Proceedings of the 7th International IEEE Symposium on High Performance Computer Architecture*, Monterrey, Mexico, Jan. 2001.

[4] FIPS 180-1. Secure Hash Standard. Technical Report Publication 180-1, Federal Information Processing Standard (FIPS), National Institute of Standards and Technology, US Department of Commerce, Washington D.C., Apr. 1995.

[5] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics,*

Speech, and Signal Processing, volume 3, Seattle, WA, May 1998.

[6] N. Ganapathy and C. Schimmel. General purpose operating system support for multiple page sizes. In *Proceedings of the USENIX 1998 Annual Technical Conference*, Berkeley, CA, June 1998.

[7] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[8] Imagemagick. http://www.imagemagick.org.

[9] G. B. Kandiraju and A. Sivasubramaniam. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Marina del Rey, CA, June 2002.

[10] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Upper Saddle River, NJ, 1992.

[11] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, Apr. 1992.

[12] Y. A. Khalidi, M. Talluri, M. N. Nelson, and D. Williams. Virtual memory support for multiple page sizes. In *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, Napa, CA, Oct. 1993.

[13] J. C. Mogul. Big memories on the desktop. In *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, Napa, CA, Oct. 1993.

[14] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421-431, June 1977.

[15] J. Poskanzer. thttpd – tiny/turbo/throttling HTTP server. http://www.acme.com/software/thttpd/.

[16] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita, Italy, June 1995.

[17] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the 15th Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[18] R. L. Sites and R. T. Witek. *Alpha Architecture Reference Manual*. Digital Press, Boston, MA, 1998.

[19] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath. Implementation of multiple pagesize support in HP-UX. In *Proceedings of the USENIX 1998 Annual Technical Conference*, Berkeley, CA, June 1998.

[20] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1994.

[21] M. Talluri, M. D. Hill, and Y. A. Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the 15th Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[22] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[23] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest, and R. Brown. Design tradeoffs for software-managed TLBs. *ACM Transactions on Computer Systems*, 12(3):175–205, Aug. 1994.

[24] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. Taylor, R. H. Katz, and D. A. Patterson. An in-cache address translation mechanism. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, Tokyo, Japan, 1986. ACM.

Vertigo: Automatic Performance-Setting for Linux

Krisztián Flautner
krisztian.flautner@arm.com
ARM Limited
110 Fulbourn Road
Cambridge, UK CB1 9NJ

Trevor Mudge
tnm@eecs.umich.edu
The University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122

Abstract

Combining high performance with low power consumption is becoming one of the primary objectives of processor designs. Instead of relying just on sleep mode for conserving power, an increasing number of processors take advantage of the fact that reducing the clock frequency and corresponding operating voltage of the CPU can yield quadratic decrease in energy use. However, performance reduction can only be beneficial if it is done transparently, without causing the software to miss its deadlines. In this paper, we describe the implementation and performance-setting algorithms used in Vertigo, our power management extensions for Linux. Vertigo makes its decisions automatically, without any application-specific involvement. We describe how a hierarchy of performance-setting algorithms, each specialized for different workload characteristics, can be used for controlling the processor's performance. The algorithms operate independently from one another and can be dynamically configured. As a basis for comparison with conventional algorithms, we contrast measurements made on a Transmeta Crusoe-based computer using its built-in LongRun power manager with Vertigo running on the same system. We show that unlike conventional interval-based algorithms like LongRun, Vertigo is successful at focusing in on a small range of performance levels that are sufficient to meet an application's deadlines. When playing MPEG movies, this behavior translates into a 11%-35% reduction of mean performance level over LongRun, without any negative impact on the framerate. The performance reduction can in turn yield significant power savings.

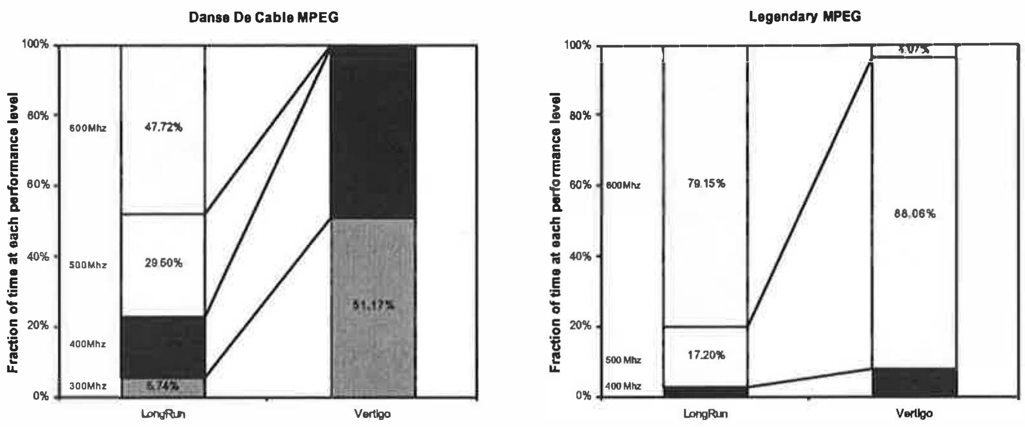
1. Introduction

Power considerations are increasingly driving processor designs from embedded computers to servers. Perhaps the most apparent need for low-power processors is for mobile communication and PDA devices. These devices are battery operated, have small form factors and are increasingly taking up computational tasks that in the past have been performed by desktop computers. The next generation 3G mobile phones promise always-on connections, high-bandwidth mobile data access, voice recognition, video-on-demand services, video conferencing and the convergence of today's multiple standalone devices—MP3 player, game machine, camera, GPS, even the wallet—into a single device. This requires processors that are capable of high performance and modest power consumption. Moreover, to be power efficient, the processors for the next generation communicator need to take advantage of the highly variable performance requirements of the applications they are likely to run. For example an MPEG video player requires about an order of magnitude higher performance than an MP3 audio player but optimizing the processor to always run at the level that accommodates the video player would be wasteful.

Dynamic Voltage Scaling (DVS) exploits the fact that the peak frequency of a processor implemented in CMOS is proportional to the supply voltage, while the amount of dynamic energy required for a given workload is proportional to the square of the processor's supply voltage [12]. Running the processor slower means that the voltage level can also be lowered, yielding a quadratic reduction in energy consumption, at the cost of increased run time. The key to making use of this trade-off are performance-setting algorithms that aim to reduce the processor's performance level (clock frequency) only when it is not critical to meeting the software's deadlines. The key observation is that often the processor is running too fast. For example, it is pointless from a quality-of-service perspective to decode the 30 frames of a video in half a second, when the software is only required to display those frames during a one second interval. Completing a task before its deadline is an inefficient use of energy [6].

While dynamic power currently accounts for the greatest fraction of a processor's power consumption, static power consumption, which results from the leakage current in CMOS devices, is rapidly increasing. If left unchecked, in a 0.07 micron process, leakage power could become comparable to the amount of dynamic power [3]. Similarly to dynamic power, leakage can also be substantially reduced if the processor does not always have to operate at its peak performance level. One technique for accomplishing this is adaptive reverse body biasing (ABB), which combined with dynamic voltage scaling can yield substantial reduction in both leakage and dynamic power consumption [11]. The pertinent point for this paper with respect to DVS and ABB is that lowering the speed of the processor results in better than linear energy savings. Vertigo provides the main lever for controlling both of these tech-

FIGURE 1. **MPEG video playback LongRun vs. Vertigo**



niques by providing an estimate for the necessary performance level of the processor.

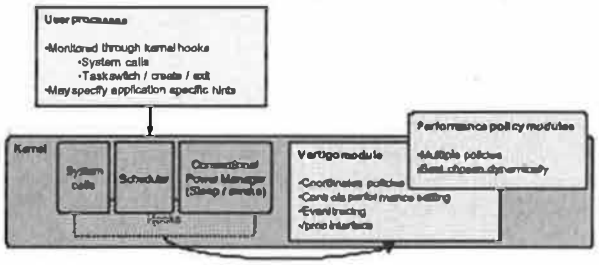
Most mobile processors on the market today already support some form of voltage scaling; Intel calls its version of this technology SpeedStep [8]. However, due to the lack of built-in performance-setting policies in current operating systems, the computers based on these chips use a simple approach that is driven not by the workload but by the usage model: when the notebook computer is plugged in a power outlet the processor runs at a higher speed, when running on batteries, it is switched to a more power efficient but slower mode. Transmeta's Crusoe processor sidesteps this problem by building the power management policy—called LongRun—into the processor's firmware to avoid the need to modify the operating system [20]. LongRun uses the historical utilization of the processor to guide clock rate selection: it speeds up the processor if utilization is high and decreases performance if utilization is low. Unlike on more conventional processors, the power management policy can be implemented on the Crusoe relatively easily because it already has a hidden software layer that performs dynamic binary translation and optimizations. However, it is currently an open question—one that we address in this paper—how effectively a policy implemented at such a low level in the software hierarchy can perform.

Research into performance-setting algorithms can be broadly divided into two categories: ones that use information about task deadlines in real-time kernels to guide the performance-setting decisions of the processor [9][13][15][19][16][17], and others that seek to derive deadlines automatically by either monitoring past utilization of the processor (interval-based techniques) [6][14][21] or based on semantic task and event classification [4][10]. Our work falls into the latter category. Previously, we presented a mechanism for automatically classifying machine utilization into different types of episodes [5] and automatically assigning deadlines to them [4]. Deadline and classification information is derived from communication patterns between the executing tasks based on observations in the OS kernel. Vertigo is built on the high-level ideas that were described in our previous papers and moves these techniques out of the simulator into a hardware and software implementation.

Our performance-setting algorithms, described in Section 2, compare favorably to previous interval-based algorithms. The two key differences in our approach are that multiple performance-setting algorithms are used to come up with a global prediction and that the algorithms are implemented in the OS kernel, which gives them access to a richer set of data for predictions. The multiple performance-setting algorithms in the system ensure that they do not all have to be optimal in all possible circumstances. This allows at least some of the algorithms to be less concerned about the worst case. Figure 1 illustrates the fraction of time spent at each of the processor's four performance levels (300, 400, 500, and 600 Mhz) using the Crusoe's built-in LongRun power manager in contrast with Vertigo during playbacks of two MPEG movies. The data for both algorithms were collected on the same hardware, however during the Vertigo measurements, the built-in LongRun power manager was disabled. While the playback quality of the different runs were identical, the main difference between the results is that Vertigo spends significantly more time below peak performance than LongRun. During the first movie, Vertigo switches mostly between two performance levels: the machine's minimum 300 Mhz and 400 Mhz, while during the second, it settles on the processor's third performance level at 500 Mhz. LongRun, on the other hand, during both movies chooses the machine's peak performance setting for the dominant portion of execution time.

Vertigo is implemented as a set of kernel modules and patches that hook into the Linux kernel to monitor program execution and to control the speed and voltage levels of the processor (Figure 2). One of the main design objectives of this system has been to be minimally intrusive into the host operating system. Vertigo coexists with the existing scheduler, system calls, and power manager (which controls the sleep and awake modes of the processor), however it needs certain hooks within these subsystems. A unique feature of

FIGURE 2. Vertigo architecture


User processes
•Monitored through kernel hooks
•System calls
•Task switch / create / exit
•May specify application specific hints
Kernel
System calls
Scheduler
Conventional Power Manager (Sleep / awake)
Hooks
Vertigo module
•Coordinates policies
•Controls performance setting
•Event tracing
•/proc interface
Performance policy modules
•Multiple policies
•Best chosen dynamically

Vertigo is that instead of a single performance-setting algorithm, it allows the composition of multiple algorithms, all specializing in different kinds of run-time situations. The most applicable to a given condition is chosen at run-time. The different performance-setting policies are coordinated by the core module, which connects to the hooks in the kernel and provides shared functionality to the policies. The shared functionality includes an abstraction for setting the processor's performance level, measuring and estimating work and a low-overhead soft-timer implementation built on timestamp counters that provides sub-millisecond resolution. Implementation issues are discussed in Section 3.

Instead of estimating the potential energy savings resulting from our techniques, we use raw performance levels as the metric of interest in this paper. The correlation between performance levels and dynamic power consumption of processors has been clearly established in the literature [9][12][13][17]. We believe that performance-setting techniques are applicable more broadly than just for controlling dynamic voltage scaling and that they will also be useful for controlling leakage-power reduction techniques in the near future [11]. However, process details for useful estimates of energy are not yet available, and current predictions are likely to be inaccurate. Evaluations of our algorithms are presented in Section 4.

The main contributions of this paper are a set of kernel-level algorithms for performance-setting under Linux, a technique for coordinating multiple algorithms, a description of the Vertigo performance-setting framework, an evaluation of our algorithms on a Crusoe-based hardware platform, and a technique for measuring and contrasting our results with the processor's built-in power manager. While Vertigo's perspectives-based algorithm is a new addition, the interactive algorithm has been described in our previous work and evaluated on a simulator [4].

2. Performance-setting algorithms

Unlike previous approaches, Vertigo includes multiple performance-setting algorithms that are coordinated to find the best estimate for the necessary performance level. The various algorithms are organized into a decision hierarchy, where algorithms closer to the top have the right to override the choices made at lower levels. Currently we have three levels on the stack:

- *At the top:* an algorithm for automatically quantifying the performance requirements of interactive applications and which ensures that the user experience does not suffer. This algorithm is based on our previous one described in [4].
- *In the middle:* an application specific layer, where DVS-aware applications can submit information about their performance requirements.
- *At the bottom:* an algorithm that attempts to estimate the future utilization of the processor based on past information. This *perspectives-based* algorithm differs from previous interval-based algorithms in that it derives a utilization estimate for each task separately and adjusts the size of the utilization-history window on a per-task basis. Moreover, since the algorithm in the top layer ensures the high quality of interactive performance, the baseline algorithm does not have to be conservative about the size of the utilization-history window, the consideration of which has led to inefficient algorithms for even simple workloads (e.g. MPEG playback) in the past [7].

In this paper our focus is on the *interactive* algorithm at the top of the stack and the *perspectives-based* algorithm at the bottom. The application-specific layer is currently only used for debugging: we have instrumented certain applications such as the X server and our mpeg player to submit application specific information to Vertigo (through a system call) and then this information can be used to correlate Vertigo's activities with that of the applications.

2.1 Keeping track of work

The main measure used in our performance-setting algorithms is the full-speed equivalent work done during an interval. This measure can be used to estimate how long a given workload would take running at the peak performance of a processor. On the Crusoe, the full-speed equivalent work estimate is computed by the formula:

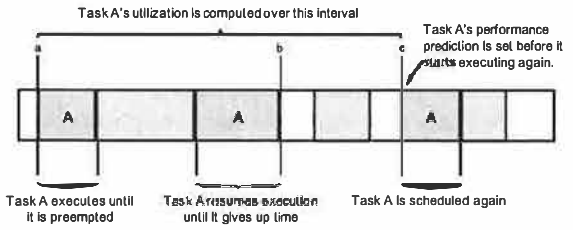
$$Work_{fse} = \sum_{i=1}^{n} t_i p_i \qquad \text{(EQ 1)}$$

Where i refers to one of the n different performance levels during a given interval with the corresponding amount of non-idle time spent at that performance level (t_i) in seconds and frequencies (p_i) specified as a fraction of peak performance. On a system where the count rate of the timestamp counter (a.k.a. cycle counter) varies with the speed of the processor, the full-speed equivalent speed would be computed differently. For example, if the timestamp counters count at the current rate of the processor, $Work_{fse}$ would simply be given as the difference between the value of the timestamp counter at the beginning and end of an interval. It is implicit in the above equation that a workload's runtime is linearly related to the inverse of the processor's performance level. However, this is not always the case, primarily due to the non-linear bus and processor speed ratios during performance-scaling. Section 3.2 deals with this issue in more detail.

2.2 A perspectives-based algorithm

At the lowest level in the policy stack is an algorithm that aims to derive a rough approximation for the

FIGURE 3. Measuring the utilization for task A


Task A's utilization is computed over this interval
a
b
c
Task A's performance prediction is set before it starts executing again.
A
A
A
Task A executes until it is preempted
Task A resumes execution until it gives up time
Task A is scheduled again

necessary performance level of the processor. It need not be completely accurate, since the assumption is that algorithms at higher positions on the policy stack will override its decisions when necessary. We refer to this algorithm as *perspectives-based*, since it computes performance predictions from the perspective of each task and uses the combined result to control the performance-setting of the processor. This algorithm differs from previous interval-based algorithms in that it derives a utilization estimate for each task separately and adjusts the size of the utilization-history window on a per-task basis.

Our insight is that individual tasks (or groups of tasks) often have discernible utilization periods at the task level, which can be obscured if all tasks are observed in the aggregate. We use each task's observed period for recomputing per-task exponentially decaying averages of the work done during the period and of its estimated deadlines. While previous interval-based performance-setting techniques also use exponentially decaying averages, they use them globally and with fixed periods. The update period in these techniques is usually set to between 10ms and 50ms, which often proves to be too short and usually causes the predictions to oscillate between only two performance levels. Their problem is that since a single algorithm must accurately set the performance level in all cases, it cannot wait long enough to smooth out the performance prediction without unduly impacting the interactive performance. Our current technique uses a simple heuristic for finding a task's period: the algorithm tracks the time from when a task starts executing, through points when it is preempted and eventually runs out of work (gives up time on its own), until the next time it is rescheduled. We have experimented with more complicated techniques for finding a task's period, such as tracking communications between them and tracking system calls [4], however we found that this simpler strategy works sufficiently well.

Figure 3 illustrates the execution of a hypothetical workload on the processor. At *point a* task A starts execution and the per-task data structures are initialized with four pieces of information: the current state of the work counter, the current state of the idle time counter, the current time, and a *run bit* indicating that the task has started running. The counters are used to compute the task's utilization and subsequently its performance requirements—see Section 3.2 for more information about how these are used. When the task is preempted, the task's run bit is left as-is, indicating that the task still has work left over. When task A gets scheduled again, it runs until it gives up time willingly (runs to completion before its schedule quantum expires or calls a system call that yields the processor to another task) at *point b* and its run bit is cleared. At *point c*, when task A is rescheduled, the cleared state of the run bit indicates that there is enough information for computing the task's performance requirements and setting the processor's performance level accordingly. At point c, $Work_{fse}$ is computed for the range between *point a* and *point c* and a future work estimate is derived based on this value (Equation 2):

$$WorkEst_{new} = \frac{k \times WorkEst_{old} + Work_{fse}}{k+1} \qquad \text{(EQ 2)}$$

A separate exponentially decaying average is maintained to keep track of the deadlines of each interval, where the deadline is computed as $Work_{fse} + Idle$, where *Idle* specifies the amount of idle time during the interval between *points a* and *c* (Equation 3):

$$Deadline_{new} = \frac{k \times Deadline_{old} + Work_{fse} + Idle}{k+1} \qquad \text{(EQ 3)}$$

Given these two values the performance-level prediction is computed as follows:

$$Perf = \frac{WorkEst}{Deadline} \qquad \text{(EQ 4)}$$

By keeping track of the work and deadline predictions separately, the performance predictions are weighted by the length of the interval over which the work estimates were measured. Note that unlike previous approaches, in this algorithm the performance predictions are used directly to set the machine's performance level, not indirectly to scale the processor's performance level up or down by an arbitrary amount [7]. Similarly to the data presented in [14], we found that small weight values for k work well, and used $k=3$ in our measurements.

As a result of our strategy, work estimates for each task are recomputed on a varying interval with a mean of around 50-150ms (depending on workload), however, as a result of multiple tasks running in the system, there is actually a refinement of the work estimate every 5ms to 10ms. One pitfall of the perspectives-based algorithm is that if there is a new non-interactive, CPU-bound task that gets started on an idle system, and that task utilizes the processor without being preempted for a long duration of time, there might be significant latency incurred in responding to the load. To guard against this situation, we put a limit on the non-preempted duration over which the work estimate is computed. If a task does not yield the processor for 100ms, its work estimate is recomputed. The 100ms value was selected based on two observations: a separate algorithm for interactive applications ensures that they meet a more stringent deadline, and that the only class of applications affected by the choice of the 100ms limit are the computationally intensive batch jobs (such as compilation) which are likely to run for seconds or minutes, and where an extra tenth of a second of execution time is unlikely to be significant.

2.3 Interactive applications

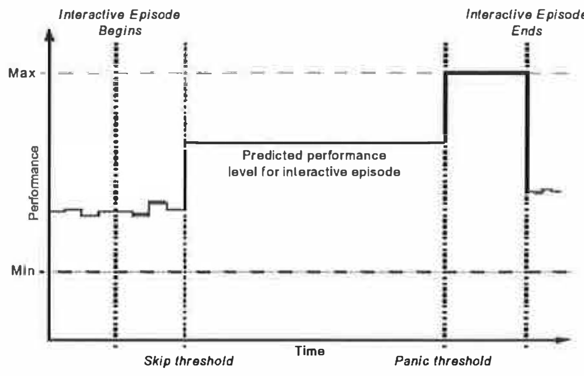
Our strategy for ensuring good interactive performance relies on finding the periods of execution that directly impact the user experience and ensuring that these episodes complete without undue delay. We use a relatively simple technique for automatically isolating interactive episodes that relies on monitoring communication from the X server (or other control task in charge of user interaction) and tracking the execution of the tasks that get triggered as a result. The technique used in Vertigo is based on our previous descriptions in [4] and [5]. A summary follows below:

The beginning of an interactive episode is initiated by the user and is signified by a GUI event, such as pressing a mouse button or a key on the keyboard. As a result of such an event, the GUI controller (X server in our case) dispatches a message to the task that is responsible for handling the event. By monitoring the appropriate system calls (various versions of read, write, and select), Vertigo can automatically detect the beginning of an interactive episode. When the episode starts, both the GUI controller and the task that is the receiver of the message are marked as being in an interactive episode. If tasks of an interactive episode communicate with unmarked tasks, then the as yet unmarked tasks are also marked. During this process, Vertigo keeps track of how many of the marked tasks have been preempted. The end of the episode is reached when that number is zero.

Figure 4 illustrates the strategy for setting the performance level during an interactive episode. At its beginning, the algorithm waits for a specific amount of time, determined by the *skip threshold* before transitioning to the predicted performance level. We observed that the vast majority of interactive episodes are so short (sub millisecond) as to not warrant any special consideration. These short episodes are the results of echoing key presses to the window or moving the mouse across the screen and redrawing small rectangles. We found that a skip threshold of 5ms is good value for filtering short episodes without adversely impacting the worst case. If the episode exceeds the skip threshold, the performance level is switched to the interactive performance prediction. Similarly to the perspectives-based algorithm, the prediction for the interactive episodes is computed as the exponentially decaying average of the correct settings of past interactive episodes. To bound the worst case impact on the user experience, if the interactive episode does not finish before reaching the *panic threshold*, the processor's performance is ramped up to its maximum. At the end of the interactive episode, the algorithm computes what the correct performance-setting for the episode should have been and this value is incorporated into the exponentially moving average for future predictions. An added optimization is that if the panic threshold was reached during an episode, the moving average is rescaled so that the last performance level gets incorporated with a higher weight (*k*=1 is used instead of *k*=3).

The performance prediction is computed for all episodes that are longer than the *skip threshold*. If the episode was also longer than *perception threshold*, then

FIGURE 4. Performance-setting for interactive episodes


Interactive Episode Begins
Interactive Episode Ends
Max
Min
Performance
Predicted performance level for interactive episode
Skip threshold
Time
Panic threshold

the performance requirement is set to 100%. The perception threshold describes a cut-off point, under which events appear to happen instantaneously for the user. Thus, completing these events any faster would not have any perceptible impact on the observer [2]. While the exact value of the perception threshold is dependent on the user and the type of task being accomplished, a value of 50ms is commonly used [2][4][14]. Equation 5 is used for computing the performance requirements of episodes that are shorter than the perception threshold.

$$Perf = \frac{Work_{fse}}{PerceptionThreshold} \qquad \text{(EQ 5)}$$

Where the full-speed equivalent work is measured from the beginning of the interactive episode. The algorithm in Vertigo differs on the following points from our previous interactive algorithm:

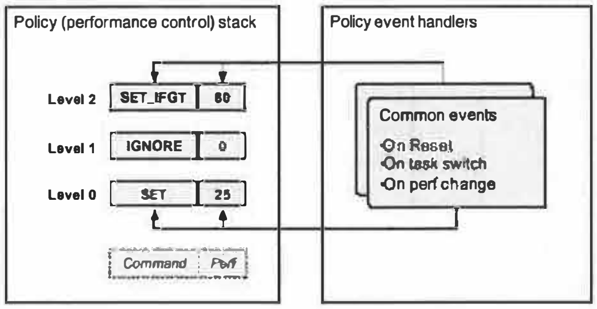
- Finding the end of an interactive episode has been simplified. We found that the higher accuracy inherent in our previous implementations was unnecessary.
- The *panic threshold* has been statically set to 50ms. In our previous implementations the threshold varies dynamically depending on the rate that work is getting done (i.e. the performance level during the interactive episode). While this idea might still prove to be useful on machines with a wider range of performance levels, we saw no perceptible difference on our evaluation machine which has a performance range of 300Mhz to 600Mhz.
- There is only a singe prediction for the necessary performance level for an interactive episode in the system. In our previous technique, we used a per-task value depending on which task initiated the episode.

3. Implementation issues

3.1 Policy stack

The policy stack (Figure 5) is a mechanism for supporting multiple independent performance-setting policies in a unified manner. The primary reason for having multiple policies is to allow the specialization of performance-setting algorithms to specific situations instead of having to make a single algorithm perform

FIGURE 5. **Policy stack**



well under all conditions. The policy stack keeps track of commands and performance-level requests from each policy and uses this information to combine them into a single global performance-level decision when needed.

The different policies are not aware of their positions in the hierarchy and can base their performance decisions on any event in the system. When a policy re quests a performance level it submits a command along with its desired performance to the policy stack. The command specifies how the requested performance should be combined with requests from lower levels on the stack: it can specify to ignore (*IGNORE*) the request at the current level, to force (*SET*) a performance level without regard to any requests from below, or to only set a performance level if the request is greater than anything below (*SET_IFGT*). When a new performance level request arrives, then the commands on the stack are evaluated bottom up to compute the new global performance level. Using this system, performance requests can be submitted any time and a new result computed without explicitly having to invoke all the performance-setting policies.

While policies can be triggered by any event in the system and they may submit a new performance request at any time, there are a set of common events that all policies tend to be interested in. On these events, instead of recomputing the global performance level each time a policy modifies its request, the performance level is computed only once after all interested policies' event handlers have been invoked. Currently the set of common events are: reset, task switch, task create, and performance change. The performance change event is a notification which is sent to each policy and does not usually cause any changes to the performance requests on the stack.

3.2 Work tracking

Our algorithms use the processor's utilization history over a given interval to estimate the necessary speed of the processor in the future. The idea is to maximize the busy time of the processor by slowing it down to the appropriate performance level. To aid this, Vertigo provides an abstraction for tracking the work done during a given time interval which takes performance changes and idle time into account regardless of the specific hardware counter implementations. To get a work measurement over an interval, a policy needs to allocate a `vertigo_work` struct and call the `vertigo_work_start` function at the beginning, and the `vertigo_work_stop` function at the end of the interval. During the measurement, the contents of the structs are updated automatically to reflect the amount of idle time and the utilized time weighted by the corresponding performance levels of the processor. This information can then be used to compute the result of Equation 1, which then can be used for performance-level prediction.

Aside from the convenience that this abstraction provides for policy writers, it is also designed to simplify porting of Vertigo (and associated policies) to different hardware architectures. One major difference between platforms is how time is measured. Many architectures provide a low overhead way of counting cycles through timestamp counters, others may only provide externally programmable timer interrupts for the user. Moreover, even when timestamp counters are provided, they do not always measure the same things. On current Intel Pentium and ARM processors the timestamp counters count cycles—the rate varies depending on the speed of the processor—and the counter stops counting when the processor transitions into sleep mode. The Crusoe's implementation of the timestamp counter measures time: it always counts the cycles at the peak rate of the processor and continues to do so even when the processor is asleep. Ideally a system would include both types of counters, however, Vertigo can be made to work with either approach.

One aspect of systems the work estimate does not yet take into account is that a workload running at half of peak performance does not necessarily run twice as long as the original. One reason for this is that as the core is slowed down, the memory system is not, thus the core to memory performance ratio improves in memory's favor [7]. Table 1 shows our measurements which show the difference between the expected and measured lengths of the workloads based on runs at 300, 400, and 500 Mhz settings of the processor. On the CPU bound loop, the difference between the predictions and actual measurements are in the noise, while on the MPEG workload, there is about a 6%-7% inaccuracy increase per 100 Mhz step. While the maximum inaccuracy on these workloads is less than 20%, as the range of minimum to maximum performance increases, along with a reduction in the range of each performance step, a more accurate work estimator might be necessary. A possible solution could be to take the instruction mix of the workload into account by the use of performance monitoring counters that keep track of significant events such as external memory accesses.

3.3 Monitoring, timers and tracing

One design goal of Vertigo has been to make it as autonomous from other units in the kernel as possible. Another design goal emerged as we selected the platform for our experiments. The Transmeta Crusoe processor includes its own performance-setting algorithm and we wished to compare the two approaches. The first requirement has already yielded a relatively unobtrusive design, the second focused us on turning the existing functionality into a passive observation platform.

TABLE 1. Scaling error of work predictions

	CPU bound loop			MPEG video		
	400 Mhz	500 Mhz	600 Mhz	400 Mhz	500 Mhz	600 Mhz
300 Mhz	-0.3%	-0.4%	-0.3%	7.1%	13.5%	19.4%
400 Mhz		-0.1%	0.0%		6.9%	13.3%
500 Mhz			0.1%			6.8%

TABLE 2. Timer statistics

Cost of an access to a timestamp counter	30-40 cycles
Mean delta between timer checks	~0.1 ms
Timer accuracy	~1 ms
Avg. timer check and dispatch duration (incl. possible execution of an event handler)	100-150 cycles

An example of how Vertigo has been made unobtrusive is the way timers are handled. Vertigo provides a sub-millisecond resolution timer, without changing the way Linux's built-in 10ms resolution timer works. This is accomplished by piggybacking the timer dispatch routine, which checks for timer events onto often-executed parts of the kernel, such as the scheduler and system calls. Since Vertigo already intercepts certain system calls to find interactive episodes and is also invoked on every task switch, it was straight-forward to add a few instructions to these hooks to manage timer dispatches. Each hook is augmented with a read of the timestamp counter, a comparison against the next timer event's time stamp and a branch to the timer dispatch routine upon success. In practice we found that this strategy yields a timer with sub-millisecond accuracy, while its worst case resolution is bound by the scheduler's time quantum, which is 10ms (see Table 2). However, since the events that Vertigo is interested in measuring usually occur close to the timer triggers, this technique has adequate resolution. Another advantage is that since the soft-timers stop ticking when the processor is in sleep mode, the timer interrupts do not change the sleep characteristics of the running OS and applications. Our technique is similar to soft-timers [1], where based on similar requirements to ours, high resolution and low overhead timers are applied to network processing.

All these features allowed us to develop, in addition to the active mode where Vertigo is in control, a passive mode, where the built-in LongRun power manager is in charge of performance-setting and Vertigo is simply an observer of the execution and performance changes. Monitoring the performance changes caused by LongRun is accomplished similarly to the timer dispatch routine. Vertigo periodically reads the performance level of the processor through a machine specific register (msr) and compares the result to its previous value. If they are different, then the change is logged in a buffer. Vertigo includes a tracing mechanism that retains a log of significant events in a kernel buffer that is exposed through the proc file system. This log includes performance-level requests from the different policies, task preemptions, task ids, and the performance levels of the processor. Another feature of this technique is that it allows us to compare LongRun and Vertigo during the same run: LongRun is in control of performance-setting while Vertigo outputs the decisions that it would have made on the same workload. We use this technique to contrast the differences between unrepeatable runs of interactive benchmarks between the two policies (see Section 4.2).

To get a better feel for the overhead of using our measurement and performance-setting techniques, Vertigo was instrumented with markers that keep track of the time spent in Vertigo code at run-time. While the run-time overhead on a Pentium II is less than 0.1% to 0.5%, on the Transmeta Crusoe it is between 1% and 4%. Further measurements in virtual machines such as VMWare and user-mode-linux (UML) confirmed that the overhead can be significantly higher in virtual machines than on traditional processor architectures. We believe that Vertigo's overhead could be reduced further since we as yet use unoptimized algorithms.

4. Evaluation

Our measurements were performed on a Sony Vaio PCG-C1VN notebook computer using the Transmeta Crusoe 5600 processor running at 300Mhz to 600Mhz with 100Mhz steps. The operating system used is Mandrake 7.2 with a modified version of the Linux 2.4.4-ac18 kernel. The workloads used in the evaluation are the following: Plaympeg SDL MPEG player library [18], Acrobat Reader for rendering PDF files, Emacs for text editing, Netscape Mail and News 4.7 for news reading, Konqueror 1.9.8 for web browsing, and Xwelltris 1.0.0 as a 3D tetris-like game. The interactive shell commands benchmark is a record of a user doing miscellaneous shell operations during a span of about 30 minutes. To avoid variability due to the Crusoe's dynamic translation engine, most benchmarks were run at least twice to warm up the dynamic translation cache, and data was used from the last run.

4.1 Multimedia

MPEG video playback poses a difficult challenge for performance-setting algorithms. While the algorithm puts a periodic load on the system, the performance requirements can vary depending on the frame's type. Thus, if a performance-setting algorithm looks at too-long of a past history for predicting future requirements, it can miss the deadlines for more computationally intensive frames. On the other hand, if the algorithm looks at only a short interval, then it will not settle on a single performance value but oscillate between multiple settings. This issue is exposed in [7], where the authors show that no heuristic algorithm they looked at could successfully settle on the single performance level that would have been adequate for the entire workload. Our observations of LongRun confirm this behaviour.

Vertigo deals with this problem by relying on the interactive performance-setting algorithm at the top of the hierarchy to bound worst-case responsiveness (in

TABLE 3. Application-level statistics about the plaympeg benchmark playing various movies

		Execution statistics			MPEG decode	
		Length (s)	Idle	Sleep	Ahead (s)	Exactly on time
Danse De Cable	LongRun	247.1	54%	23%	148.10	6
320x160 +audio	Vertigo		27%	4%	68.74	1012
Legendary	LongRun	19.4	33%	13%	7.20	19
352x240 +audio	Vertigo		24%	7%	4.79	65
Red's Nightmare	LongRun	49.1	48%	36%	26.31	5
320x240	Vertigo		32%	13%	16.53	74
Red's Nightmare	LongRun	49.3	22%	15%	12.48	87
480x360	Vertigo		18%	11%	8.17	139
Roadkill Turtle	LongRun	121.3	46%	19%	64.93	5
304x240 +audio	Vertigo		25%	4%	33.34	237
Sentinel	LongRun	35.6	28%	10%	11.05	80
320x240 +audio	Vertigo		19%	5%	6.32	231
SpecialOps	LongRun	60.8	30%	11%	19.01	129
320x240 +audio	Vertigo		20%	5%	12.67	305

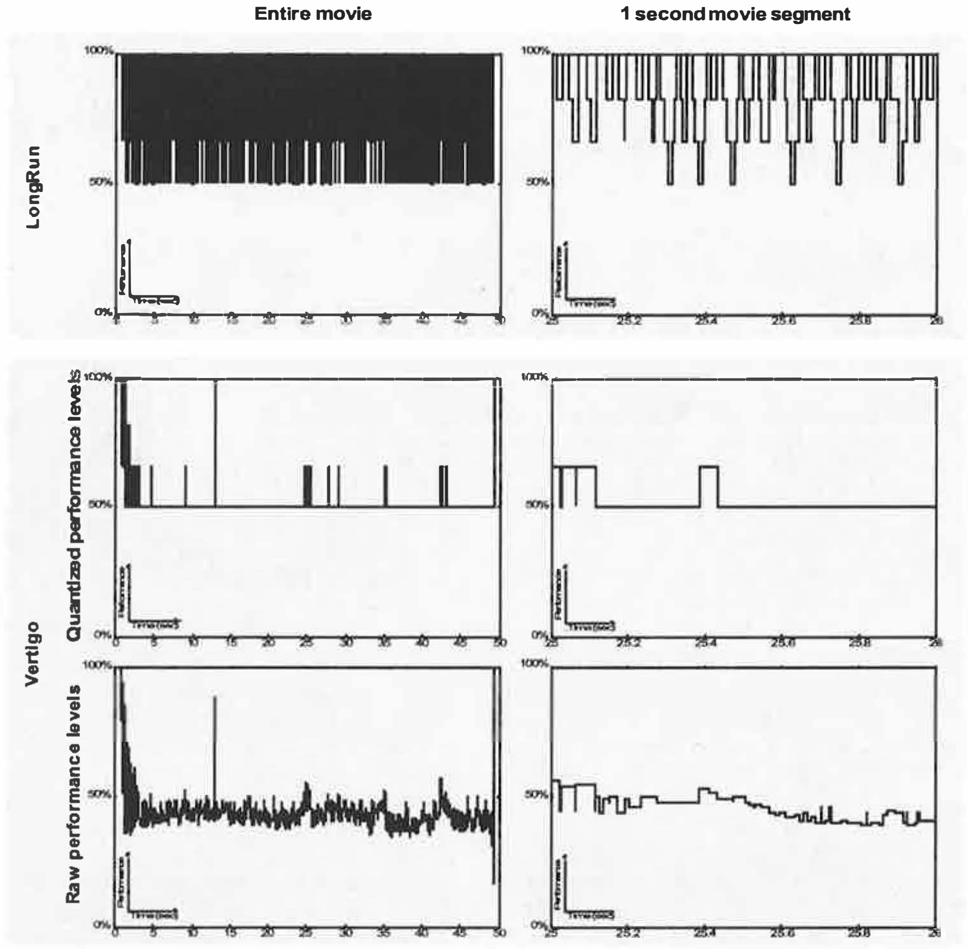
this case frame rate) and allowing the more conventional interval-based algorithm at the bottom of the hierarchy to take a longer-term view. Table 3 shows measurements for the plaympeg video player [18] playing a variety of MPEG videos. Some of the internal variables of the video player have been exposed to provide information about how the player is affected as the result of changing the processor's performance levels during execution. These figures are shown in the *MPEG decode* column of the table. The *Ahead* variable measures how close the end of each frame's decoding is to its deadline. It is expressed as cumulative seconds during the playback of each video. For power efficiency, this number should be as close to zero as possible, although the slowest performance level of the processor puts a limit on how much its value can be reduced. The *Exactly on time* field specifies the number of frames that met their deadlines exactly. The more frames are on time, the closer the performance-setting algorithm is to the theoretical optimum. The data in the *Execution Statistics* column is collected by Vertigo's monitoring subsystem. To collect information about LongRun, Vertigo was used in passive mode to gather a trace of performance changes without controlling the processor's performance level. The difference between the *Idle* and *Sleep* fields are that the first corresponds to the fraction of time spent in the kernel's idle loop—possibly doing housekeeping chores or just spinning—while the latter shows the fraction of time the processor actually spends in a low-power sleep mode.

Table 4 provides statistics about the processor's performance levels during the runs of each workload.

TABLE 4. Performance levels during movie playback

	LongRun					Vertigo					Mean performance reduction over LongRun
	Fraction of time at each performance level (Mhz)				Mean perf level	Fraction of time at each performance level (Mhz)				Mean perf level	
	300	400	500	600		300	400	500	600		
Danse De Cable	6%	19%	33%	**54%**	89%	**51%**	48%	0%	0%	59%	**34%**
Legendary	0%	3%	17%	**79%**	96%	0%	8%	**88%**	4%	82%	**15%**
Red's Nightmare small	11%	**35%**	**35%**	19%	80%	**95%**	2%	0%	3%	52%	**35%**
Red's Nightmare big	0%	5%	21%	**74%**	95%	0%	0%	**90%**	10%	85%	**11%**
Roadkill Turtle	3%	10%	23%	**64%**	92%	1%	**97%**	1%	0%	66%	**28%**
Sentinel	0%	0%	14%	**86%**	97%	0%	0%	**93%**	7%	84%	**13%**
SpecialOps	1%	2%	14%	**83%**	96%	0%	2%	**93%**	4%	83%	**14%**

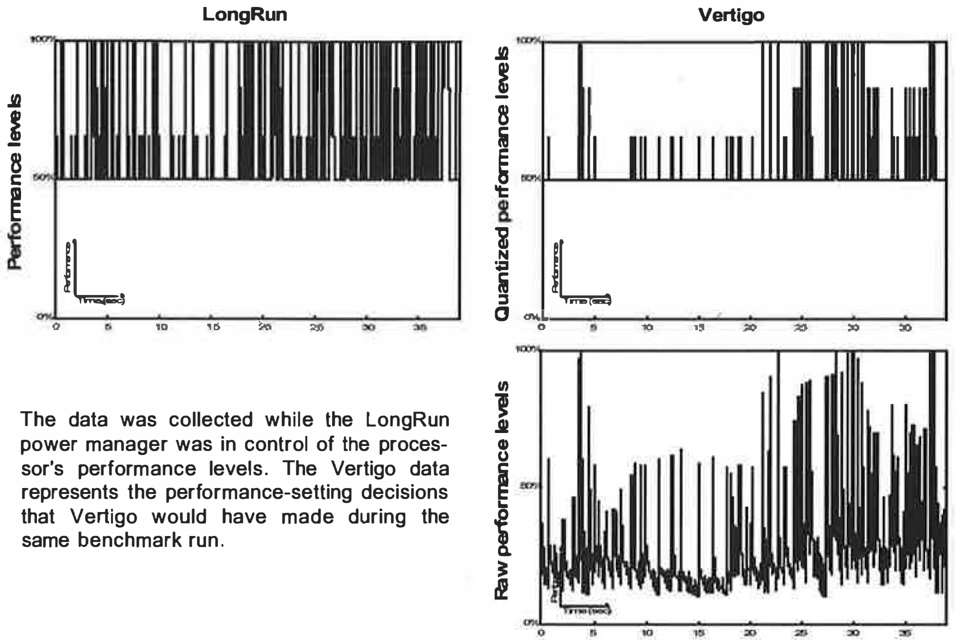
FIGURE 6. Performance-setting during MPEG playback of Red's Nightmare 320x240



The fraction of time at each performance level is computed as a proportion of total non-idle time during the run of the workload. The *Mean perf level* column specifies the average performance levels (as the percentage of peak performance) during the execution of each workload. Since, in all cases, the mean performance level for each workload was lower using Vertigo, the last column specifies the amount of reduction. The playback quality for each pair of workloads was the same: same frame rate and no dropped frames. Our results show that Vertigo is more accurately able to predict the necessary performance level than LongRun. The increased accuracy results in a 11% to 35% reduction of the average performance levels of the processor during the benchmarks' execution. Since the amount of work between runs of a workload stays the same, the lower average performance level implies reduced idle and sleep times when Vertigo is enabled. This expectation is affirmed by our results. Similarly, the number of frames that exactly meet their deadlines increases when Vertigo is enabled and the cumulative amount of time when decode is ahead of its deadline is reduced. The median performance level (highlighted with bold in each column) also shows significant reductions. While on most benchmarks Vertigo settles on a single performance level below peak for the greatest fraction of execution time (>88%), LongRun usually chooses to run the processor at full throttle. The exception to this is the Danse De Cable workload, where Vertigo settles on the lowest two performance levels and switches between the two continuously. The reason for this behavior is due to the specific performance levels on the Crusoe processor; Vertigo would have wanted to select a performance level which is only slightly higher than 300 Mhz and as the prediction fluctuates below and above that value, it is quantized to the closest two performance levels.

The biggest single difference between LongRun and Vertigo is that LongRun appears to be overcautious: it ramps up the performance level very quickly when it detects significant amounts of processor activity. Over all workloads, the average performance level with LongRun never gets below 80%, while Vertigo goes down as low as 52%. Vertigo is less cautious but responds quickly when the quality of service appears to have been compromised. Since LongRun does not have any information about the interactive performance, it is forced to act conservatively on a shorter time frame, which leads to inefficiencies. Figure 6 provides qualitative insight into the characteristics of the two different performance-setting policies. LongRun keeps on ramp-

FIGURE 7. **Performance-setting decisions during the Konqueror benchmark**



The data was collected while the LongRun power manager was in control of the processor's performance levels. The Vertigo data represents the performance-setting decisions that Vertigo would have made during the same benchmark run.

ing the performance level up and down in fast succession, while Vertigo stays close to a target performance level. The top row shows the processor's performance levels during a benchmark run with LongRun enabled and the bottom two rows show the same benchmark for Vertigo. The middle row shows the actual performance levels during execution, while the bottom row reflects the performance level that Vertigo would request on a processor that could run at arbitrary performance levels (given the same max. performance). Note that in some cases, Vertigo's desired performance levels are actually below the minimum that's achievable on the processor.

4.2 Interactive workloads

Due to the difficulty in making interactive benchmark runs repeatable, interactive workloads are significantly harder to evaluate than the multimedia benchmarks. To get around this problem, we combined empirical measurements with a simple simulation technique. The idea is to run our benchmarks under the control of the native LongRun power manager and only engage Vertigo in passive mode, where it merely records the performance-setting decisions that it would have made but does not actually change the processor's performance levels. Figure 7 shows the performance data that was collected during a run of our measurements. The LongRun graph corresponds to the actual performance levels of the processor during the measurement, while the Vertigo graphs show the quantized performance levels that it would have used had it been in control. Note that if Vertigo were in control, its performance-setting decisions would have had a different run-time impact from LongRun, thus the time axis on the Vertigo graph are only approximations.

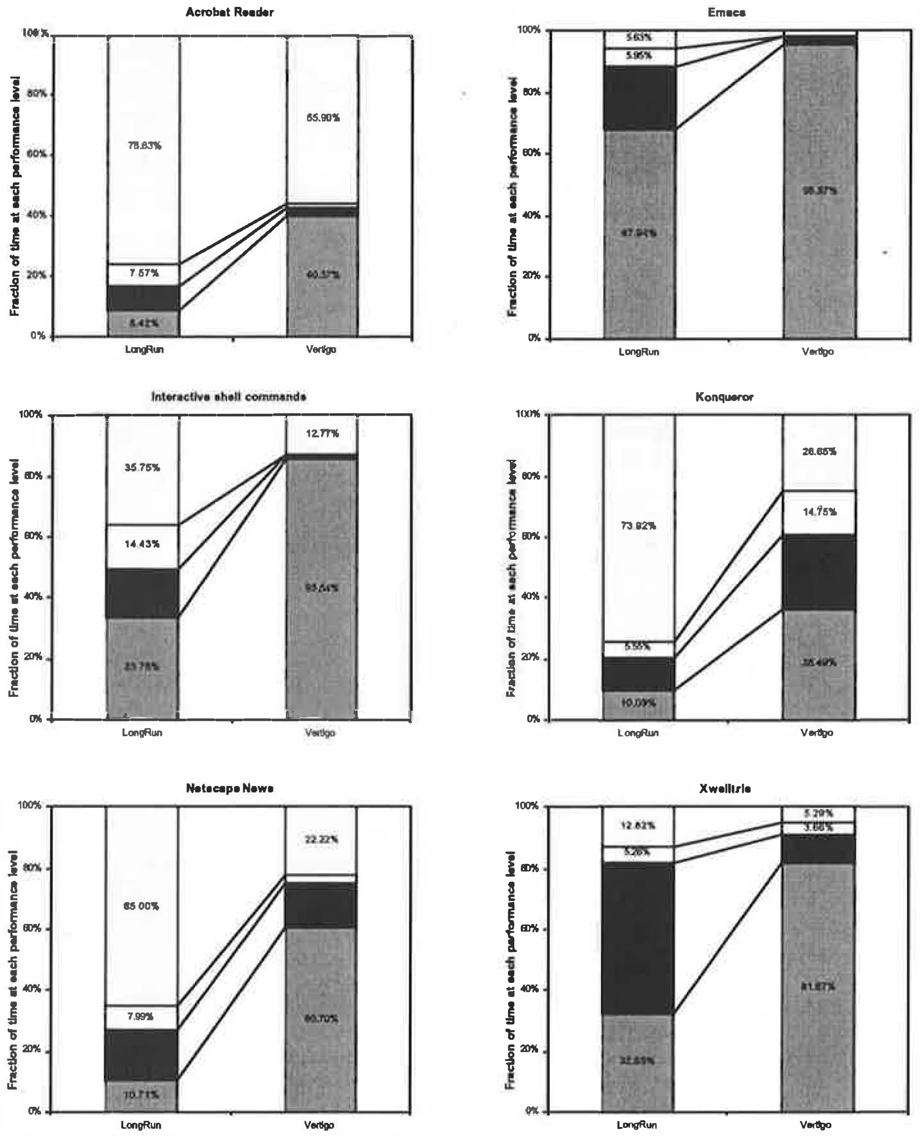
To get around the time-skew problem in our statistics, the passive Vertigo performance-level traces were postprocessed to take the impact of the increased execution times that would have resulted from the use of Vertigo instead of LongRun. Instead of looking at the entire performance-level trace, we chose to focus only on the interesting parts: the interactive episodes. As part of the interactive performance-setting algorithm in Vertigo, it includes a technique for finding durations of execution that have a direct impact on the user. This technique gives valid readings regardless of which algorithm is in control and is used to focus our measurements. Once the execution range for an interactive episode has been isolated, the full-speed equivalent work done during the episode is computed for both LongRun and Vertigo. Since during the measurement LongRun is in control of the CPU speed and it runs faster than Vertigo, the latter's episode duration must be lengthened. First, the remaining work is computed for Vertigo (Equation 6). Then, the algorithm computes how much the length of the interactive episode needs to be stretched—assuming that Vertigo continues to run at its predicted speed until reaching the panic threshold, at full-speed after that—and the statistics are adjusted accordingly.

$$Work_{VertigoRemaining} = Work_{LongRun} - Work_{Vertigo} \quad \text{(EQ 6)}$$

We found that the results using this technique are close to what we observed on similar workloads (same benchmark but with slightly different interactive load) running with Vertigo active. However, when Vertigo is in control, the number of performance-setting decisions are reduced and are more accurate.

Figure 8 shows the statistics gathered using the above technique. Each graph contains two stacked columns, corresponding to the fraction of time spent in interactive episodes at each of the four performance levels supported in our computer. These performance levels—from bottom up—are from 300 Mhz to 600 Mhz at 100 Mhz increments. Even from a high level, it

FIGURE 8. Fraction of time at different performance levels



is apparent that Vertigo spends more time at lower performance levels than LongRun. On some benchmarks such as Emacs, there is hardly ever a need to go fast and the interactive deadlines are met while the machine stays at its lowest possible performance level. On the other end of the spectrum is Acrobat Reader, which exhibits bimodal behaviour: the processor either runs at its peak level or at its minimum. Even on this benchmark many of the interactive episodes can complete in time at the machine's minimum performance level, however when it comes to rendering the pages, the peak performance level of the processor is not sufficient to complete its deadlines under the user's perception threshold. Thus, upon encountering a sufficiently long interactive episode, Vertigo switches the machine's performance level to its peak. On the other hand, during the run of the Konqueror benchmark, Vertigo can take advantage of all four performance levels that are available on the machine. This is in contrast with LongRun's strategy which causes the processor to spend most of its time at the peak level.

5. Conclusions and future work

We have shown how two performance-setting policies implemented at different levels in the software hierarchy behave on a variety of multimedia and interactive workloads. We found that Transmeta's LongRun power manager, which is implemented in the processor's firmware, makes more conservative choices than our Vertigo algorithms, which are implemented in the Linux kernel. On a set of multimedia benchmarks, the different design decisions result in a 11%-35% average performance level reduction by Vertigo over LongRun. Being higher on the software stack allows Vertigo to make decisions based on a richer set of run-time information, which translates into increased accuracy. While the firmware approach was shown to be less accurate than an algorithm in the kernel, it does not diminish its usefulness. LongRun has the crucial advantage of being operating system agnostic. Perhaps one way to bridge the gap between low and high level implementations is to provide a baseline algorithm in firmware and expose an interface to the operating system to optionally refine performance-setting decisions. The policy stack in Vertigo can be viewed as the beginnings of a mechanism to support such design, where the bottom-most policy on the stack could actually be implemented in the processor's firmware.

We believe that aside from dynamic voltage scaling, performance-setting algorithms will be useful for controlling other power reduction techniques, such as adaptive body biasing. These circuit techniques cut down on the processor's leakage power consumption, which is an increasing fraction of total power as the feature sizes of transistors are reduced. While the power consumption of the processor is a significant concern, it only accounts for a fraction of the system's total power consumption. Future work will extend our technique to managing the power of all the devices in an integrated system.

6. Acknowledgements

Thanks to Deborah Wallach and our shepherd, Carla Ellis for their comments and suggestions on draft versions of this paper. This work was supported in part by DARPA grant F33615-00-C-1678 and NSF grant CCR-0073497.

References

[1] M. Aron and P. Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. *Proceedings of the 17th Symposium on Operating Systems Principles (SOSP-17)*, December 1999.

[2] S. K. Card, T. P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates, Publishers, 1983.

[3] A. Chandrakasan, W. Bowhill, F. Fox eds., *Design of High-Performance Microprocessor Circuits*. Piscataway, NJ: IEEE Press, 2001.

[4] K. Flautner, S. Reinhardt, and T. Mudge. Automatic Performance-Setting for Dynamic Voltage Scaling. *Proceedings of the International Conference on Mobile Computing and Networking (MOBICOM-7),* July *2001.*

[5] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism and interactive performance of desktop applications. *Proceedings of Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), November 2000.*

[6] K. Govil, E. Chan, and H. Wasserman. Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU. *Proceedings of the First International Conference on Mobile Computing and Networking*, November 1995.

[7] D. Grunwald, P. Levis, K. Farkas, C. B. Morrey III, and M. Neufeld. Policies for Dynamic Clock Scheduling. *Proceedings of the Fourth Symposium on Operating Systems Design & Implementation*, October 2000.

[8] Intel SpeedStep. http://support.intel.com/support/processors/mobile/pentiumiii/ss.htm

[9] C. M. Krishna and Y-H Lee. Voltage-Clock-Scaling Adaptive Scheduling Techniques for Low Power Hard Real-Time Systems. *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000), 2000.*

[10] J. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. *Proceedings of the ACM SIGMETRICS 2001 Conference*, June 2001.

[11] S. Martin, K. Flautner, D. Blaauw, and T. Mudge. Combined Dynamic Voltage Scaling and Adaptive Body Biasing for Optimal Power Consumption in Microprocessors under Dynamic Workloads. *Proceedings of the International Conference on Computer Aided Design (ICCAD 2002)*, San Jose, CA, November 2002.

[12] T. Mudge. Power: A First Class Architectural Design Constraint. IEEE Computer, vol. 34, no. 4, April 2001.

[13] T. Okuma, T. Ishihara, and H. Yasuura. Real-Time Task Scheduling for a Variable Voltage Processor. *Proceedings of the International Symposium on System Synthesis*, November 1999.

[14] T. Pering, T. Burd, and R. Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. *Proceedings of International Symposium on Low Power Electronics and Design 1998*, pp. 76-81, June 1998.

[15] T. Pering, T. Burd, and R. Brodersen. Voltage Scheduling in the lpARM Microprocessor System. *Proceedings of the International Symposium on Low Power Electronics and Design 2000*, July 2000.

[16] P. Pillai and K. G. Shin. Real-time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. *Proceedings of the 18th Symposium on Operating System Principles*, October 2001.

[17] J. Pouwelse, K. Langendoen, and H. Sips. Voltage scaling on a low-power microprocessor. *Proceedings of the International Conference on Mobile Computing and Networking (MOBICOM-7),* July *2001.*

[18] SDL MPEG player library. http://www.lokigames.com/development/smpeg.php3

[19] Y. Shin and K. Choit. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. *Proceedings of the 36th Annual Design Automation Conference, 1999.*

[20] Transmeta Crusoe. http://www.transmeta.com/technology/index.html

[21] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. *Proceedings of the First Symposium of Operating Systems Design and Implementation*, November 1994.

Cooperative I/O—A Novel I/O Semantics for Energy-Aware Applications

Andreas Weissel, Björn Beutel, Frank Bellosa

University of Erlangen, Department of Computer Science
`{weissel,bnbeutel,bellosa}@cs.fau.de`

Abstract

In this paper we demonstrate the benefits of application involvement in operating system power management. We present Coop-I/O, an approach to reduce the power consumption of devices while encompassing all levels of the system—from the hardware and OS to a new interface for cooperative I/O that can be used by energy-aware applications. We assume devices which can be set to low-power operation modes if they are not accessed and where switching between modes consumes additional energy, e.g. devices with rotating components or network devices consuming energy for the establishment and shutdown of network connections. In these cases frequent mode switches should be avoided.
With Coop-I/O, applications can declare open, read and write operations as *deferrable* and even *abortable* by specifying a time-out and a cancel flag. This information enables the operating system to delay and batch requests so that the number of power mode switches is reduced and the device can be kept longer in a low-power mode. We have deployed our concept to the IDE hard disk driver and Ext2 file system of Linux and to typical real-life programs so that they make use of the new cooperative I/O functions. With energy savings of up to 50%, the experimental results demonstrate the benefits of the concept. We will show that Coop-I/O even outperforms the "oracle" shutdown policy which defines the lower bound in power consumption if the timing of requests can not be influenced.

1 Introduction

Mobile devices and embedded systems characterize one major trend in computer system design. One common aspect of new developments in this area is a limited power supply and the need to economically handle the energy resource. Energy-aware system design has therefore been widely recognized to be an important and major challenge [19].

To enable *dynamic power management*, the control and reduction of power consumption at run time, many techniques and algorithms have been suggested. Research in this area has focused mainly on the operating system level. The OS tries to save a maximum amount of energy by setting devices and components to low-power modes if they are not accessed. Because a mode switch itself consumes energy, this action will only pay off if the time to the next request is long enough. An introduction to low-power modes of devices is given in section 2.

Our contribution to this research area is *Coop-I/O*—an approach that integrates the application layer into dynamic power management of devices. We introduce a new operating system interface for cooperative I/O which can be exploited by energy-aware applications. File operations are equipped with two additional parameters—a time-out and a cancel flag. The operating system tries to batch deferrable requests in order to create long idle periods during which switching to a low-power mode pays off.

An example of a deferrable and abortable write operation is the periodic auto-save function of a text editor. If an auto-save has to be aborted because the disk is shut down, the next auto-save can be performed non-cooperatively with up-to-date data. Deferrable, but not abortable read operations could fill the read buffer of an audio- or video player. The time-out can be set to the play time of this buffer. Other examples are background processes like cron jobs, daemons or logging mechanisms. A web browser could use a memory cache and abortable reads and writes to access its disk cache. If the disk is not running, data will be cached only in memory and not on disk.

Our concept consists of three major parts, which will be presented in detail in the next sections:

- We introduce new cooperative file operations that have a time-out and a cancel flag as additional parameters (see section 3). If a file operation needs to access a disk drive and that drive is shut down, the operation will be suspended until either the disk drive has spun up due to another I/O request or the time-out has elapsed. When the time-out is reached and the file operation cancel flag is set, the operation will be aborted. In all other cases, it will be finally executed. The new functions are compatible with the legacy interface.
- The operating system caches disk blocks in *block buffers* in main memory. Modified block buffers are periodically written to disk by an update mechanism. We present an update policy that is redesigned to save energy (see section 4).
- The operating system will control the hard disk modes; it will switch the disk drive to a low-power mode when it assumes that the drive will not be accessed for a certain time. This is controlled by a simple algorithm called *device-dependent time-out* (DDT) (see section 5).

Figure 1 presents the whole concept. Coop-I/O integrates all levels—hardware, operating system (driver, cache and file system) and the application layer—to reduce energy consumption

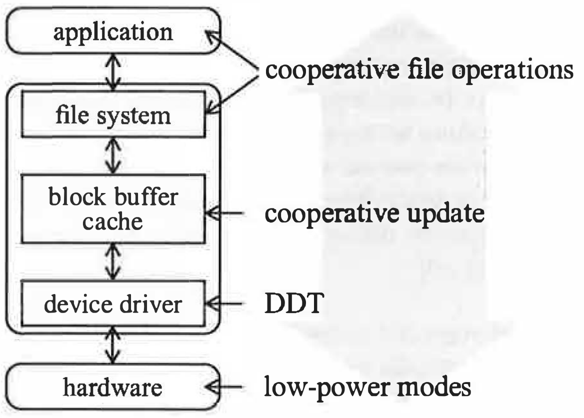


Figure 1: components of Coop-I/O

We have implemented Coop-I/O in Linux. The extensions to the IDE device driver, the buffer management and the file system are presented in section 6. As we will show, only little effort is needed to rewrite existing programs and energy-savings are possible even if many unmodified applications are running. The results of our experiments can be found in section 7. We will show that even the "oracle" device shutdown policy, which has complete knowledge about, but no influence on the timing of future requests, can be surpassed by Coop-I/O. Related work is presented in section 8.

2 Low-Power Modes of Devices

Modern peripheral devices make use of several operation modes that are associated with different levels of power consumption. Operating systems may reduce the power consumption of a device by switching between the device's operation modes.

The ATA standard, also known as IDE, defines four operation modes for hard disks [1]: *active, idle, standby* and *sleep*.

- *Active mode:* the hard disk is reading, writing, seeking or spinning up/down.
- *Idle mode:* the disk is rotating and the interface is active. The head is moved to a parking position.
- *Standby mode:* the hard disk spindle motor is off, but the hard disk interface is still active.
- *Sleep mode:* both the hard disk spindle motor and the interface are off. To reactivate, a reset command is needed.

A device is "shut down" if it is put into standby or sleep mode. A non-negligible amount of time and energy is needed to enter and leave these modes. Therefore, entering standby or sleep mode will only pay off if the interval to the next disk operation is long enough. The minimum interval between two disk operations for which switching pays off is called the *break-even time*. It only depends on the characteristics of the hard disk device. A time interval of idleness (an idle period) that is longer than the break-even time is called a *standby period*.

The break-even time of the IBM hard disk we used in our tests is 8.7 s (see section 7.2). Note that this value is device-specific. Lu et al. [15] report break-even times of 6.39 s for a Fuijitsu MHF 2043AT disk and 35.0 s for an Hitachi DK23AA-60 drive.

3 Cooperative File Operations

Contemporary operating systems serve read requests immediately and buffer write requests before storing the data on disk according to a time-out policy. This can cause a hard disk to spin up even if the disk was shut down only a few seconds before. The application developer has no possible way to exert influence on this mechanism. With Coop-I/O, applications can soften this strict timing to enable a flexible schedule of device requests.

The classical device interface of operating systems has been designed to hide aspects regarding hardware management. The application programmer is not aware of power-saving techniques inside the operating system. With Coop-I/O we abandon this concept to some extent. We enable the application programmer to support the operating system's efforts to save energy. The details concerning the power management algorithm are still hidden from the application layer.

The essential file operations in most operating systems are provided by the system calls `open()`, `read()` and `write()`. The operations `close()` and `lseek()` usually do not access the disk directly, but operate on data in main memory. So we introduce three cooperative variants: `open_coop()`, `read_coop()` and `write_coop()`. The legacy interface, `open()`, `read()` and `write()`, is mapped to the cooperative functions with zero time-out and inactive cancel flag.

The user-specified time-out value indicates when the operation is initiated at the latest, not when the operation has finished. As with the classical Unix file I/O interface the user does not know when the operation will be completed.

3.1 Interactions between the disk cache and cooperative operations

For efficiency reasons, modern operating systems do not serve write requests to disk drives immediately. To reduce the number of slow disk operations, they keep copies of disk blocks (called *block buffers*) in a cache in main memory. A write operation copies data into block buffers which are marked as *dirty* to indicate that they differ from the blocks on disk. The *dirty buffer life span* determines the time when these buffers must be written back to disk to prevent data loss in case of a crash. A special *update* task periodically writes out dirty buffers with an elapsed life span.

Cooperative read operations. If a cooperative read references a disk block that is not cached in memory, the operation will have to check if the corresponding hard disk is in active or idle mode. If it is, the read request can be served immediately. If not, the operation will have to block itself until either the hard disk spins up due to another request or until the time-out has elapsed. If the time-out has elapsed and the *cancel flag* is set, the operation will have to be aborted. Otherwise the drive has to be activated.

Cooperative write operations. As we never modify disk blocks directly (data is always written to the block buffer cache) there seems to be no need for cooperative writes. The update task defers write operations until their dirty buffer life span has elapsed. We can easily modify this mechanism to be cooperative and wait for other device accesses (read operations); see section 4.2.

However, writing to a disk block can induce a read operation if the block is not yet cached and must be read before it can be modified. Thus a shut-down drive would have to run up immediately. In this case, a cooperative write operation will simply wait for the drive the same way as the `read_coop()` operation does.

But the situation is more complicated: if the write operation needs to read an uncached block *after* several modifications of cached blocks and the whole operation has to be cancelled (because the drive is in standby mode, the time-out has elapsed and the operation is declared abortable), we have to undo all previous modifications of cached blocks to assure file system consistency. All modifications issued by a write request can be understood as *one* transaction that must be performed completely (*commit*) or not at all (*abort*).

We decided to use the following approach which avoids the implementation of an undo mechanism: the *early commit/abort* strategy decides to commit or abort as soon as the first modification to a block buffer is going to take place.

Assume we want to modify a buffer and the drive is in standby mode. We can distinguish three situations:

1. The drive is activated due to another request before the time-out of our request is reached. We can commit the whole operation.
2. The drive is still shut down when the time-out of our request is reached. If there are no dirty buffers for the drive, it can be concluded that our request is the only one that wants to access the drive. Depending on the cancel flag, we have to activate the drive or abort the whole operation.
3. If there is another dirty buffer for the same drive (or the buffer to be modified is already dirty), the drive will run up in the near future anyway to write back that buffer. So we can immediately modify our buffer at almost no cost: When the dirty block buffer is updated to disk, our buffer will be updated in the same sweep, as described in section 4.2.

Therefore, a write to a block buffer should be delayed only as long as the drive is in standby mode *and* there are no dirty buffers for that drive. Since a write operation's first buffer modification involves committing the operation, a write can be committed even if the hard disk is not running.

Due to the early commit/abort strategy it is conceivable that an abortable write operation will not be aborted after the time-out even if the disk is in standby mode. This is the case when a read follows a committed write to a cached block:

1. The disk is in standby mode and there exist dirty block buffers for that device.
2. The write operation has to modify several disk blocks.
3. According to the *early commit/abort* strategy the write operation is committed after the modification of the first block.
4. If a subsequent block is not cached, it has to be read from the disk. This action will be deferred until the time-out has elapsed.
5. As the complete operation is already committed, we have to spin up the disk to read the block.

Fortunately, the energy waste in this case is not that big because the hard disk would run up anyway in the near future to save the dirty buffers.

Cooperative file open operations. Opening a file results in reading its meta data (inode block etc.). If the file has to be created or to be truncated, open will cause write operations. Therefore we decided to provide a cooperative system call to open files. Read and write operations induced by `open_coop()` are mapped to their corresponding cooperative counterparts.

3.2 Using the new system calls

Many applications can be modified to be cooperative so that users will not notice changes in system behavior. Examples are low-priority tasks like cron jobs, logging mechanisms or applications with periodic I/O requests like audio and video players and voice recorders.

Only little effort is needed to make use of the new system calls. Because the cooperative reads and writes may block for the specified delay time, I/O should be decoupled from main processing. An additional thread reads or writes data cooperatively and caches it for the main thread. The main thread reads or writes data from/to this buffer in the same way as it formerly operated on the file system. The additional thread thus hides the cooperative operations from the original application. Our experiments show that the amount of changes is typically in the range of 100 to 150 lines of code.

Applications which write out temporary data and update it periodically can declare their requests as deferrable and *abortable*. An example is the autosave function of a text editor. If a write request is cancelled, the autosave thread can simply ignore it and issue another cooperative write with a time-out equal to the autosave period.

4 Cooperative update

In this section we will show how the caching mechanism can be optimized with respect to power consumption.

4.1 "Traditional" caching of disk blocks

The update process passes the modifications to the cached disk blocks on to the hard disk. In Unix systems an update takes place when one of the following conditions is met:

- An explicit update command like `sync()` forces the system to write back the buffers of a file system or a file.
- The *dirty buffer life span* has elapsed. This is the most frequent cause of writing back when there is little I/O traffic. A dirty buffer whose life span has elapsed is not written back immediately, but when it is found by the update task.
- A certain percentage of block buffers is dirty. To avoid I/O jams, some of them are written back. This is the most frequent cause of dirty buffer updates when there is heavy I/O traffic.
- The system needs main memory and writes back some dirty buffers that it will reclaim as free memory later.

This common policy is not optimized to save energy. The update task has to access the disk each time it wakes up to write back dirty buffers. If the time between two updates is shorter than the break-even time, standby periods will never be reached. In this case the disk can not be set to standby mode to save energy.

4.2 Batching write requests

In active mode, energy consumption is higher than in idle or standby mode. Furthermore, switching between modes consumes a significant amount of energy. As a consequence write requests should be batched to maximize the time the device can spend in low-power modes and to reduce mode switches.

To make updates and thus disk requests less frequent, we use a policy which updates each drive independently of all others and is preferably executed when another disk request is generated (*drive-specific cooperative update*). It is composed of four strategies:

- *Write back all buffers.* We write back all dirty buffers instead of only the oldest ones, so we have to update at most once per dirty buffer life span (60 s in our implementation).
  An additional, possibly redundant write operation will have only marginal costs if it is batched with other requests.
- *Update cooperatively.* The operating system tries to join other hard disk accesses (read requests) and write back the dirty buffers possibly before the dirty buffer life span has elapsed. This or the expiration of the full life span will trigger the update process.
  By attaching to a request that has to happen anyway, we can update at very little cost.
- *Update each drive separately.* This will not compromise file system consistency and it may increase the update interval for a single drive even more. It may also balance system I/O load since different drives will probably be updated at different times. Besides, this is a prerequisite for cooperative updates.
  For each drive, we have to watch the age of the oldest dirty buffer. If it has reached the dirty buffer life span, we will write back all buffers for that drive.
- *Update on shutdown.* If the operating system has decided to shut down a drive, it will first write back all dirty buffers that contain blocks of that drive. This minimizes the risk that the disk has to spin up again soon solely because there are some old dirty buffers that must be updated.

When an application reads some data from disk, it normally needs the data for further processing. Thus read operations are batched only if the application permits it by using `read_coop()`.

5 An adaptive shutdown policy

Many power management policies for hard disks have been suggested varying in complexity and usefulness (see the related work discussed in section 8). If the timing of hard disk requests cannot be influenced, the imaginary "*oracle*" policy is the policy that reaches maximum energy savings. It shuts down the hard disk at the beginning of every standby period and runs it up again so that it is just ready at the end of that standby period. To achieve this, the oracle policy needs information on future hard disk requests. This policy can only be simulated by analyzing request traces offline. The operating system of course is not able to perfectly predict the future and thus cannot achieve maximum reduction in energy consumption.

The spin-down policy is not central to our work because our focus lies on cooperative I/O and cache management. Thus we decided to choose a simple and easy to implement, but effective and proven, algorithm. Simple shutdown policies switch to a low-power mode after a fixed or adaptively changed time-out. As is stated in [15], the device-dependent time-out policy (DDT), which uses the break-even time of a drive as its time-out parameter, has good power-saving facilities, and its algorithm is fast, simple and storage-efficient. It is defined in the following way:

Shut down the hard disk if

$$t_{la} + t_{be} \leq t$$

The variables have the following meanings:

t: the current time

t_{la}:the time of the last hard disk access

t_{be}:the break-even time

Figure 2: the DDT policy

The hard disk driver has to keep track of t_{la}. Furthermore, it should know t_{be} for the drives under its control.

6 Prototype Implementation in Linux

We have implemented the Coop-I/O concept in the Linux kernel, version 2.4.10. The kernel modifications can be divided into three parts:

- The VFS and the Ext2 file system have been modified to support the drive-specific cooperative update policy of section 4. We have also introduced cooperative system calls using the concept of section 3 (see sections 6.1 & 6.2).
- The block device code, which is the glue between a particular block device driver and the file system, has been augmented to enable cooperation between the disk drivers' power mode control, the file system's update mechanism and the cooperative file operations.
- The IDE driver has been enhanced by a power mode control for hard disk drives, which includes the DDT algorithm of section 5. See section 6.3 for a detailed description.

6.1 Cooperative file operations

A file operation may block whenever it is going to access a disk or to make a clean block buffer dirty by modifying it. The blocking mechanism is implemented in the new function `wait_for_drive()`.

When blocked in `wait_for_drive()`, a task may be awoken by one of four events:

- *The timer has elapsed.*
  If the request should be cancelled on time-out, `wait_for_drive()` will return `-ETIME`.
- *The drive is serving another request.*
  The file operation can go on.
- *The number of dirty buffers for the drive has become non-zero.*
  If `wait_for_drive()` is also waiting for that event, it will simply return without error. If not, it will be ignored.
- *A signal has arrived.*
  The blocked file operation should be aborted with `-EINTR`, so `wait_for_drive()` returns with that error code. The cooperative operation should not use Linux's implicit restart mechanism since the signal could be sent to abort it.

The implementation of the cooperative file operations (`open_coop()`, `read_coop()`, `write_coop()`) is straightforward: The functions that implement the standard file operations have to be enhanced by the `time-out` parameter and the `cancel` flag. When a block is going to be read from disk, the function `wait_for_drive()` has to be called. For a write operation or an open operation that truncates an old file or creates a new one a point has to be found where the operation decides to commit or to abort. We chose to implement the early commit/abort strategy as described in section 3.1.

6.2 Drive specific cooperative update

We have implemented the drive-specific cooperative update policy (see section 4.2). Since the file system does not know about drives, we had to introduce a mapping of device numbers to drives as part of the file system. For each drive, the file system must also keep track of the number of dirty buffers and of the time when the oldest dirty buffer got dirty.

The update task in the original Linux wakes up every 5 s. The cooperative version of the update task also wakes up when a drive is accessed and the file system finds out that it is opportune to update that drive, as explained in section 4.2. The need for a cooperative update is checked every time a drive is read from or written to. If there are any dirty buffers for the drive and the drive's oldest dirty buffer is older than half of the dirty buffer life span, the update task will be woken up and induced to update that drive.

6.3 Power mode control for IDE drives

Drive-specific information. For each hard disk, the Linux IDE driver keeps a description that reflects the properties and state of that device. We have augmented the device structure with information needed by the DDT algorithm (break-even time and time of the last access) and a field indicating the current power mode.

The IDE power task. A power mode switch might take a rather long time, since it may write all dirty buffers back to that drive, or it may execute an IDE command that actually changes the drive's mode and wait for its completion. Instead of a mode switch function that could block for a long time, we have introduced a kernel thread called `idepower` which serves all IDE drives.

The `idepower` thread normally sleeps and waits for a semaphore that signals that a power mode change has been requested. In this case it will wake up and emits a power mode command to the hard disk. When changing the power mode, the power task also informs the file system when dirty buffers must be written back or cooperative file operations that are blocked must be awoken. Some functions like disk operations change the power mode implicitly by emitting other IDE commands, so they must inform the power task.

There are two main reasons why a new power mode might be requested:

- A hard disk request is sent to the device driver. This implicitly changes the drive's power mode to active.
- The DDT standby algorithm decides to shut down the drive.

Some special IDE commands leave the disk drive in an undefined power mode, so they request the power task to check.

Going to standby. The DDT algorithm is implemented as a timer-based function that is called once per second. Since disk requests may be very frequent, this is more efficient than using a dedicated timer for each drive that has to be restarted when a disk request has been served.

The information that is needed by the DDT algorithm (time of the last access) is updated with each disk request.

7 Experiments and Results

7.1 Data acquisition

To validate the Coop-I/O concept a power measuring environment was needed. A comparatively inexpensive four-channel analog-to-digital converter (ADC) has been designed and built to act as the data acquisition system. It measures the voltage drop at defined sense resistors in the 5 V lines leading from the power supply to the hard disk and interfaces to the standard parallel port on the data acquisition system to output the digital values. The voltage drop is acquired with a resolution of 256 steps and at a rate of up to 20000 samples per second. The maximum voltage drop that is correctly converted is 50 mV.

The target computer was a standard personal computer running Linux (kernel version 2.4.10). The system was equipped with a 2,5" IBM Travelstar 15GN (IC25N010ATDA04) hard disk [9] which was used as the test drive.

7.2 Determining hard disk parameters

We measured the power consumption of different mode switches of the IBM hard disk with our data acquisition system. The Travelstar splits up the idle mode into three submodes (*performance, active* and *low power*) that have different power consumption and timing characteristics. Observing recent user request patterns, an internal adaptive algorithm switches autonomously between these modes [9].

Figure 3 shows the power consumption of the IBM hard disk during an idle-standby-idle turnaround.

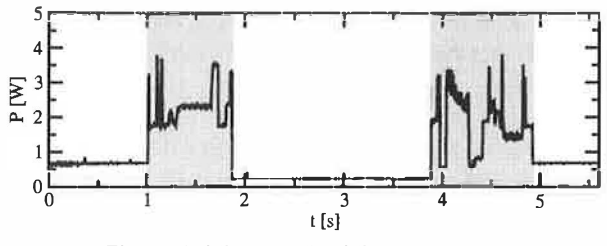


Figure 3: idle-standby-idle turnaround

t = 1s: The disk receives a shutdown command. The shaded region shows the hard disk switching from *low power idle* to *standby* mode.

t = 1.8 s: After stopping the spindle motor, the disk has reached *standby mode*, and power consumption drops to about 0.25 W.

t = 3.8s: The drive receives a write command and starts to spin up. The shaded region shows the hard disk switching from standby mode to *active mode*. Starting the spindle motor is quite expensive, energetically. After 1 s, the disk has spun up and may serve read or write requests.

t = 4.8 s: In this test scenario only a single disk block gets written. Then, the disk switches to *low power idle mode*.

We determined the following characteristics for the IBM Travelstar 15GN drive. Due to the undocumented internal adaptive algorithm of the firmware the time and energy values vary according to the recent access pattern. The following values present the average of several measurements:

time needed for a mode switch:

low-power idle to standby:	$t_{is} = 0.85$ s
standby to low-power idle:	$t_{si} = 1.03$ s

energy required to switch between modes:

low-power idle to standby:	$E_{is} = 1.91$ J
standby to low-power idle:	$E_{si} = 1.89$ J

power consumption of the low-power modes:

low-power idle mode:	$P_i = 0.657$ W
standby mode:	$P_s = 0.235$ W

break-even time: $t_{be} = 8.7$ s

7.3 Testing a cooperative audio player

We examined to what extent Cooperative I/O is able to save energy in a real-life situation. A typical application for hand-held or portable computers is a player for audio or video files. We have tested the system with a modified version of AMP, an MPEG audio layer 3 player for Linux, which makes use of the cooperative system calls. Thus we gained insight into the procedure and amount of modifications needed to make applications energy-aware.

The modified AMP creates a thread that reads from the hard disk and puts the data into a 512 kB buffer. When the player thread needs some data it reads from the buffer. The read thread and the player thread synchronize by the use of semaphores. The buffer is divided into two semi-buffers. When a semi-buffer is empty, the reader refills it by the use of a cooperative system read call while the player reads from the other semi-buffer. We had to add only about 150 source lines to incorporate the changes.

AMP was tested under the following four strategies:

- *Cooperative:*
  Use the DDT standby algorithm together with the new buffer cache and update mechanism. To read in new data, use the `read_coop()` system call with a delay that is equivalent to the playing time for one semi-buffer.
- *ECU* (*E*nergy-aware *C*aching & *U*pdate):
  Use the DDT standby algorithm together with the new buffer cache and update mechanism. Use the standard `read()` system call instead of `read_coop()` to read in new data.
- *DDT only:*
  Use the DDT standby algorithm with the *original* buffer cache and update mechanism.
- *None:*
  Do not use any power-saving measures at all.

In addition to that we simulated the "*uncooperative oracle*" policy. We collected traces of hard disk requests issued by the original uncooperative AMP running on an unmodified Linux. We calculated the minimum in total energy consumption according to the following assumptions:

- The hard disk will be set to standby mode *immediately* after serving a request if the following idle period is a standby period, i.e. if it is longer than the break-even time.
- Otherwise the hard disk is not shut down. It switches immediately to low power idle mode.

The values for "oracle" resemble the theoretical lower bounds of power consumption that can be reached by shutdown policies *without* influencing the timing of requests (in contrast to Coop-I/O).

Each strategy has been tested by playing the following two audio files. Delay is the time interval in which one semi-buffer is played. The files have the same length (9 minutes), but different compression levels.

audio file	bit rate	delay
Toccata	64 kb/s	32 s
Pastorale	128 kb/s	16 s

We have also examined how well the power-saving strategies work when an asynchronous second application runs while playing an audio file. For that aim, the test computer has concurrently executed a mail reader that examined the input mailbox of a remote computer via POP3 every minute. If there is any mail in it, the mail will be stored in the local mailbox on the test hard disk. Mail was sent in intervals of 15–60 seconds; the timing was controlled by a pseudo-random generator. For every test pass, the random generator was initialized to the same value, so the timely sequence of read/write operations was the same for each test with a tolerance of about one second. Figure 4 shows the results (all tests run for 534 seconds).

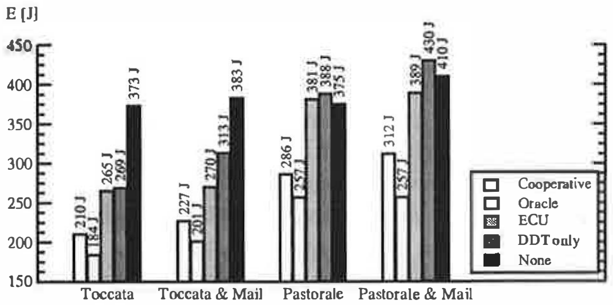
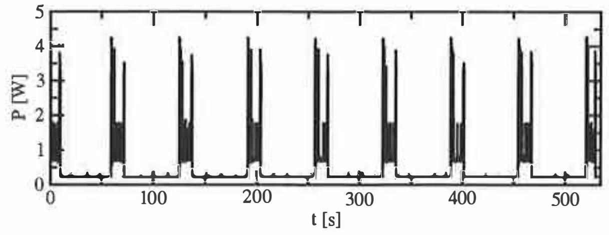


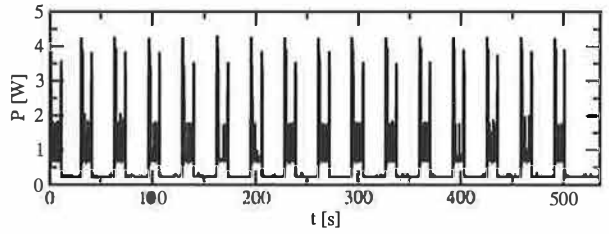
Figure 4: comparison of policies

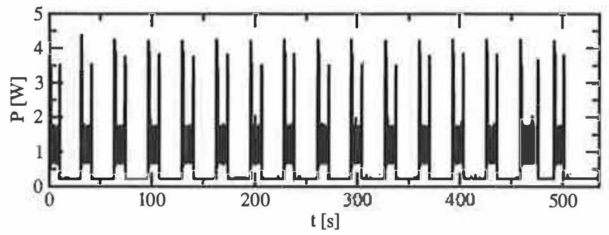
The cooperative strategy is surprisingly power-efficient in these tests. This is not only caused by the cooperation of multiple processes because some tests have only one process doing I/O. Instead, it can be explained by the following behavior: When the drive is in standby mode, a cooperative read is delayed until the data is really needed, i. e., the semi-buffer to be read will soon be played. When the delayed read operation is eventually performed, the other semi-buffer gets empty very soon and is read in immediately because the hard disk drive is still in idle or active mode. This effectively batches two subsequent read operations. You can see this behavior in figure 5.

Playing "Pastorale" together with the mail reader consumes nearly the same energy, regardless which non-cooperative strategy is used. Because the delay for this audio file is only 16 s and there are several write requests, no strategy will normally try to shut down the hard disk in this test scenario. This is due to the short intervals between requests which seldom exceed the break-even time. Coop-I/O again groups requests together, so that longer idle periods and thus *standby periods* are achieved. As a consequence the drive can be set to standby mode more often and the energy consumption is reduced.

If we have a look at the times spend in active, idle and standby mode, it can be seen that "Oracle" saves more energy than "Cooperative" by keeping the drive in standby mode all the time it is not accessed (table 1). This is due to the DDT policy which always waits 8.7 seconds before setting the drive to standby mode. The oracle policy will shut down the drive immediately if the following idle period is longer than the break-even time.


a) Cooperative (210 J)


b) Energy-aware Caching & Update (265 J)


c) DDT only (269 J)

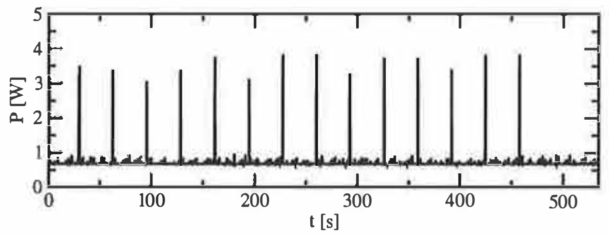

d) None (373 J)

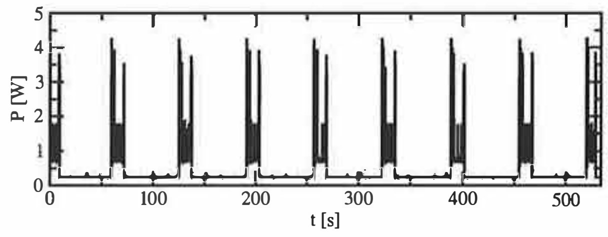
Figure 5: the four energy saving strategies (playing "Toccata", without Mail)

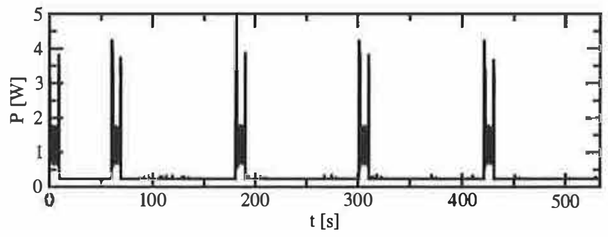
policy	active	idle	standby
Cooperative	38 s	166 s	331 s
Oracle	89 s	0 s	456 s

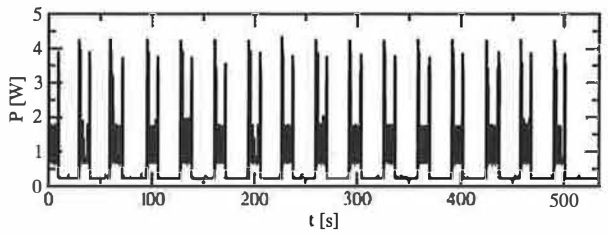
Table 1: times spent in the different power modes when playing "Pastorale"

Furthermore it can be seen that "Cooperative" reduces the time spent in active mode by almost 60%. There is almost no difference in energy consumption between the strategies "DDT only" and "None" when playing "Pastorale". This behavior is caused by the unlucky relation of the shutdown time-out and the hard disk request pattern which sets the disk to standby shortly before the disk is accessed. In that case, switching to standby mode is more expensive than staying in idle mode.

Figure 6 shows how disk requests of two independent tasks may interact. The "Toccata" task cooperatively reads one semi-buffer in every period of 32 s (figure 6a); the "Mail" task writes in intervals of 1 minute, provided that mail has arrived (figure 6b). The write requests are delayed by the cooperative update scheme (figure 6c). With Coop-I/O most of the requests of the two applications can be grouped together. As a consequence, the energy consumption of "Toccata & Mail" is only 17 J higher than without the mail application. If the requests were not coordinated, the hard disk's energy consumption would be about 60 J higher (figure 6d). This means that Coop-I/O is a working energy-saving concept.


a) Toccata cooperative (210 J)


b) Mail cooperative (164 J)


c) Toccata & Mail non-cooperative (270 J)

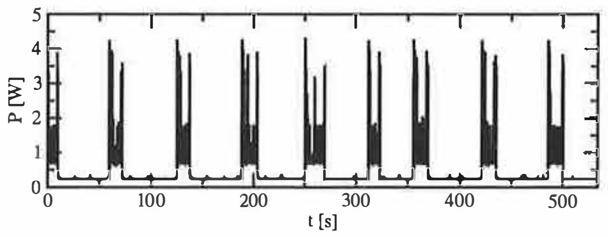

d) Toccata & Mail cooperative (227 J)

Figure 6: interaction of two independent tasks

7.4 Parameterized tests

To simulate a workload where multiple tasks periodically read or write data, we implemented two simple test programs. The period and the idle time (the time to wait at the beginning of a period until the read/write operation is started) can be configured. To simulate non-regular behavior, minimum and maximum values for the idle time can be specified; the actual value is chosen by a pseudo-random generator. Groups of five read/write processes with varying period lengths and idle times were used to generate non-regular hard disk request patterns. We measured the energy consumption for a time window of 1000 seconds (figures 7 and 8).

Reads with varying period length. In the first test series, we have varied the period length and idle times for read operations. The average period lengths of the six tests range from 25 to 150 seconds, the average idle times lie between 7 and 120 seconds. Each test has been executed in combination with the four power-saving strategies. Figure 7 shows the measured consumptions.

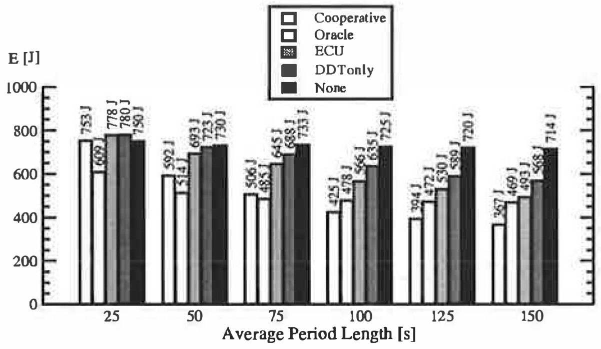


Figure 7: reads with varying average period length

Running with an average period length of 25 seconds, the energy consumption is nearly equal for all strategies (except "Oracle"). Here, the read requests are so frequent that the drive has no chance to shut down. "ECU" and "DDT only" consume even more power than "None" by initiating disadvantageous shutdowns. The longer the period length, the longer the power mode control can hold the disk in standby mode for all strategies but "None". The cooperative strategy generates the longest standby periods since following cooperative reads are delayed.

"Cooperative" even outperforms the oracle policy for average period lengths of 100 seconds and more. Table 2 shows the times spent in active, idle and standby mode for an average period length of 150 seconds. The cooperative policy batches hard disk requests. Consequently mode switches are less frequent and the time spent in active mode is reduced by more than 70%, while the time spent in low-power modes is increased. Thus "Cooperative" saves more energy than the oracle policy.

policy	active	idle	standby
Cooperative	29 s	153 s	868 s
Oracle	107 s	132 s	811 s

Table 2: times spent in the different power modes for an average period length of 150 seconds

Writes with varying period length. The energy-related characteristics of write operations are influenced by the sequence of requests and the update policy of the buffer management.

We have executed the same test series as in the previous section using write operations instead of read operations. Again, the tests have been run under all four strategies. The traces for the oracle policy where collected using the unmodified update mechanism. The results are presented in figure 8.

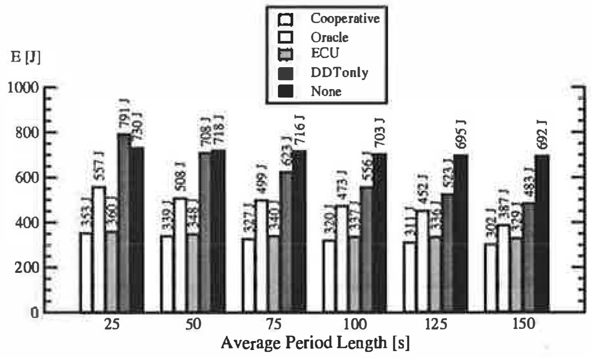


Figure 8: writes with varying average period length

The frequency of write operations seems to have little effect on the energy consumption. For the strategies "Cooperative" and "ECU" the consumption stays constantly on a low level for all period lengths. This is caused by the cooperative update policy that will write back quite regularly in intervals of 60 seconds, if no other disk request happens. The amount of data written has almost no influence on the energy consumption. The small difference between the cooperative and the "ECU" strategy indicates that cooperative write operations have a marginal but at least consistently positive benefit. The policies "DDT only" and "None" employ the original Linux 2.4 update strategy. "ECU" consumes significantly less energy than "DDT only" and "None" because of the improved update policies (see section 4.2). This shows that the original Linux update strategy does not match power-saving requirements.

7.5 Varying the number of cooperative processes

In the previous tests all programs were cooperative. But in a real-life scenario non-cooperative legacy applications will mix with cooperative programs. Thus, we have examined the behavior of a mixture of different numbers of cooperative and non-cooperative processes. Figure 9 presents the results of 6 read tests with a mixture of 5 processes including 0 to 5 cooperative applications.

The energy consumption steadily declines with increasing proportion of cooperative processes, but having a single cooperative process suffices to save energy

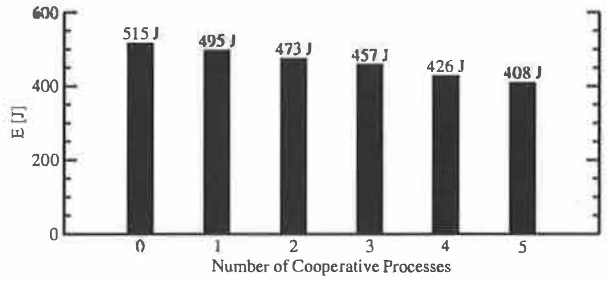


Figure 9: varying number of cooperative processes

8 Related Work

We distinguish three levels on which energy saving techniques can be applied: the hardware, the operating system and the application layer. We will show how power management is performed on each level and how Coop-I/O and existing techniques can be incorporated into an energy-aware system design.

Device-level power management. Attention has first been drawn to the lowest level leading to many research and industrial efforts at developing low-power hardware. Modern-day devices and their components can be shut down or put into low-power modes to save energy when they are not in use. These improvements have come along with OS-level support in the form of heuristic hardware-centric shutdown policies which are transparent to applications.

Three approaches to hard disk power management can be distinguished: algorithms with a fixed or adaptive time-out, policies that try to predict future requests by observing the utilization of the device and policies based on statistical models of requests. Several shutdown policies have been suggested [4],[7],[8],[10],[11],[14]. Lu et al. [15] have compared and evaluated several algorithms. Among commercial solutions to OS-level power management are the Advanced Configuration and Power Interface (ACPI) [3], Microsoft's OnNow [16] and ACPI4Linux [2].

Energy-aware operating system. As the next step more sophisticated energy-related methods have been introduced lifting energy to a first class operating system resource by unifying resource management, introducing energy accounting and enabling control of energy consumption and battery lifetime. These approaches have explicitly been designed to be not dependent on application software to be rewritten.

Zeng et al. present ECOSystem [19], a modified Linux, that unifies energy accounting over diverse hardware components and enables fair allocation of available energy among applications. ECOSystem provides system-wide energy management and can be configured to hit a specified target battery lifetime. Energy accounting is realized by implementing the powerful concept of resource containers which serve as the abstraction to which energy expenditures are charged. As Zeng et al. propose in their paper, future research should consider the interaction between ECOSystem and the applications. We find it interesting to investigate the potential of combining Coop-I/O and ECOSystem, being orthogonal to each other, to come closer to the vision of incorporating energy management in all levels of system design.

Cooperation between the OS and applications. While traditional power management schemes in operating systems do not distinguish different sources of requests, the power reduction technique introduced by Lu et al. [12] uses information about concurrently running tasks as an accurate system-level model of requesters. This *task-based power management* records the device and processor utilization of each process and shuts down a device when the overall utilization is low. Being somewhat orthogonal to each other it would be interesting to combine Coop-I/O and task-based power management.

Another approach by Lu et al. [13] is based on application involvement in energy management. New system calls are introduced which enable applications to inform the operating system about future hard disk requests. These system calls are similar to timers; they indicate in which time intervals, how often (once, periodically) and with which possible delay requests are issued. Thus applications have to inform the operating system about future requests *before* these requests are issued, normally at program start.

The cooperation policy of Coop-I/O enables processes to pass the *urgency* (delay time) of each individual request without the need for the programmer to determine request patterns of the whole program run. Figure 10a shows a scenario with four processes; number one issues several hard disk requests. While Lu's approach (Fig 10b) arranges the schedule of processes to create a cooperative schedule of hard disk requests, Coop-I/O (Fig 10c) schedules the hard disk requests themselves without changing the process schedule.

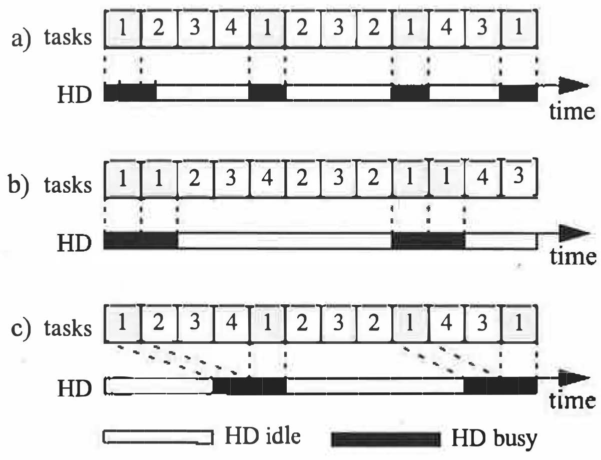


Figure 10: different request batching policies

Pouwelse et al. [18] demonstrate the benefits of power-aware applications. Applications can inform the OS about their processing demands, so the optimal processor speed can be selected that minimizes power consumption and still meets the application's deadlines.

Application adaptation. Orthogonal to our approach is *energy-aware application adaptation* presented by Flinn et al. [6],[17]. Applications are rewritten so that they can dynamically modify their behavior according to changing restrictions in energy consumption. An example of adaptation is a movie player with different quality of service modes (e.g. refresh rates or window sizes). To guide such adaptation, the operating system monitors energy supply and demand and informs applications about restrictions in energy consumption via upcalls.

Energy-aware system design. All four approaches—energy-aware hardware, operating system, application adaptation and Coop-I/O—are complementary to each other and can be used to form a comprehensive energy-aware system de-sign, as shown in figure 11.

9 Conclusion

In this paper we have presented an energy-saving concept for devices with low-power operation modes based on a cooperative relationship between the operating system and applications. While many efforts have been made to incorporate energy-related aspects in the design of operating systems and computer hardware, the interface between the OS and applications has been retained unchanged to make power management fully transparent to applications.

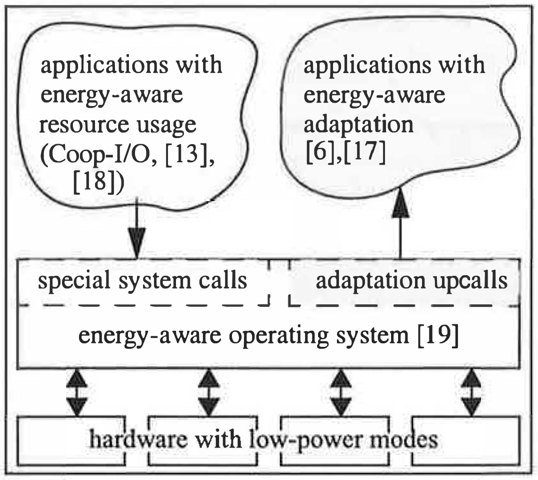

applications with energy-aware resource usage (Coop-I/O, [13], [18])
applications with energy-aware adaptation [6],[17]
special system calls
adaptation upcalls
energy-aware operating system [19]
hardware with low-power modes

Figure 11: comprehensive energy-aware system design

We have demonstrated that Coop-I/O, with its new interface functions, enables a higher reduction in power consumption than power management techniques which operate solely on the OS level. Coop-I/O outperforms the "oracle" policy which defines the theoretical lower bound in power consumption if the timing of requests is not influenced. These improvements come with the cost of modifying existing applications. We have shown that only little effort is needed to make use of the new functionality. Coop-I/O is the first approach to power management which controls not only the timing but also the execution of requests.

We plan to investigate the applicability of our concept to the control of other system components, e.g. a wireless network adapter. Our modifications to the operating system interface presented here concentrate on the basic file I/O operations. We plan to investigate to what extent other interface functions can be enhanced so that a wider range of energy-aware applications is able to contribute to OS power management. In our opinion the research area of application involvement in operating system power management facilities shows huge potential and should be further investigated.

Acknowledgments

The anonymous reviewers and our shepherd, Carla Ellis, have helped us to improve this paper with their useful feedback.

References

[1] American National Standards Institute. Information Technology – AT Attachment with Packet Interface 5 (ATA/ATAPI-5). Published as ANSI/INCITS 340-2000, Dec 2000

[2] M. Berger, S. Richter, ACPI4Linux. http://phobos.fs.tum.de/acpi/index.html

[3] Compaq, Intel, Microsoft, Phoenix, Toshiba. Advanced Configuration and Power Interface Specification 2.0a, Mar 2002

[4] F. Douglis, P. Krishnan, B. Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In *Proceedings of the Second USENIX Symposium on Mobile and Location Independent Computing*, Apr 1995

[5] F. Douglis, P. Krishnan, B Marsh. Thwarting the Power Hungry Disk. In *Proceedings of the 1994 Winter USENIX Conference*, Jan 1994

[6] J. Flinn, M. Satyanarayanan: Energy-aware Adaptation for Mobile Applications, In *Proceedings of the 17th Symposium on Operating Systems Principles SOSP'99*, pp. 48–63, Dec 1999

[7] P. Greenawalt. Modeling Power Management for Hard Disks. In *Proceedings of the Symposium on Modeling and Simulation of Computer and Telecommunication Systems*, Jan 1994

[8] D. Helmbold, D. Long, B. Sherrod. A Dynamic Disk Spin-Down Technique for Mobile Computing. In *Proceedings of the 2nd ACM International Conference on Mobile Computing (MOBICOM96)*, pp. 130–142, Nov 1996

[9] IBM Corporation. Hard Disk Drive Specifications for Travelstar 48GH, 30GN & 15GN, Rev. 2.0, Jan 2002

[10] P. Krishnan, P. Long, J. Vitter. Adaptive Disk Spin-Down via Optimal Rent-to-Buy in Probabilistic Environments. In *Proceedings of the 12th International Conference on Machine Learning*. pp. 332–330, July 1995

[11] Li-K; Kumpf-R; Horton-P; Anderson-T. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *Proceedings of the USENIX Winter 1994 Conference*, pp. 279–292, Jan 1994

[12] Y.-H. Lu, L.Benini, G. De Micheli. Operating System Directed Power Reduction. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design ISLPED'00*, pp. 37–42, July 2000

[13] Y.-H. Lu, L. Benini, G. De Micheli. Power-aware Operating Systems for Interactive Systems, In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(2):119–134, Apr 2002

[14] Y.-H. Lu, G. De Micheli. Adaptive Hard Disk Power Management on Personal Computers. In *Proceedings of the Ninth IEEE Great Lakes Symposium*, pp. 50–53, Mar 1999

[15] Y.-H. Lu, G. De Micheli: Comparing System-Level Power Management Policies. In *IEEE Design & Test of Computers. Special Issue on Dynamic Power Management of Electronic Systems*. pp. 10–18, Mar-Apr 2001

[16] Microsoft. OnNow Power Management http://www.microsoft.com/hwdev/onnow/

[17] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, K. R. Walker. Agile Application-aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles SOSP'97*, pp. 276–287, Oct 1997

[18] J. Pouwelse, K. Langendoen, H. Spis. Dynamic Voltage Scaling on a Low-power Microprocessor. In *Proceedings of the International Symposium on Mobile Multimedia Systems & Applications MMSA'2000*, Nov 2000

[19] H. Zeng, X. Fan, C. Ellis, A. Lebeck, A. Vahdat: ECOSystem: Managing Energy as a First Class Operating System Resource, In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2002

TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks*

Samuel Madden, Michael J. Franklin, and Joseph M. Hellerstein
{madden,franklin,jmh}@cs.berkeley.edu
UC Berkeley

Wei Hong
wei.hong@intel-research.net
Intel Research, Berkeley

Abstract

We present the Tiny AGgregation (TAG) service for aggregation in low-power, distributed, wireless environments. TAG allows users to express simple, declarative queries and have them distributed and executed efficiently in networks of low-power, wireless sensors. We discuss various generic properties of aggregates, and show how those properties affect the performance of our in network approach. We include a performance study demonstrating the advantages of our approach over traditional centralized, out-of-network methods, and discuss a variety of optimizations for improving the performance and fault-tolerance of the basic solution.

1 Introduction

Recent advances in computing technology have led to the production of a new class of computing device: the wireless, battery powered, smart sensor [25]. These new sensors are active, full fledged computers, capable not only of measuring real world phenomena but also filtering, sharing, and combining those measurements. One example of such small sensor devices are the *motes* under development at UC Berkeley. Current generation motes are roughly 2cm x 4cm x 1cm and are equipped with a radio, a processor, memory, a small battery pack, and a suite of sensors. The mote operating system, TinyOS, provides a set of primitives designed to facilitate the deployment of motes in *ad-hoc* networks. In such networks, devices can identify each other and route data without prior knowledge of or assumptions about the network topology, allowing the network topology to change as devices move, run out of power, or experience shifting waves of interference.

Due to the relative ease of deployment of mote-based sensor networks, practitioners in a variety of fields have begun considering them for a range of monitoring and data collection tasks. For example: civil engineers are using motes to monitor building integrity during earthquakes [31]; biologists are planning mote deployments for habitat monitoring[21, 5]; administrators of large computer clusters are interested in using motes to monitor the temperature and power usage in their data centers.

All of these sensor applications depend on the ability to extract data from the network. Often, this data consists of summaries (or aggregations) rather than raw sensor readings. Other researchers have noted the importance of data aggregation in sensor networks [13, 10, 12]. This previous work has tended to view aggregation as an application-specific mechanism that would be programmed into the devices on an as-needed basis, typically in error-prone, low-level languages like C. In contrast, our position is that because aggregation is so central to emerging sensor network applications, it must be provided as a *core service* by the system software. Instead of a set of extensible C APIs, we believe this service should consist of a generic, easily invoked high-level programming abstraction. This approach enables users of sensor networks, who often are not networking experts or even computer scientists, to focus on their applications free from the idiosyncrasies of the underlying embedded OS and hardware.

1.1 The TAG Approach

We have developed Tiny AGgregation (TAG), a generic aggregation service for *ad hoc* networks of TinyOS motes. There are two essential attributes of this service. First, it provides a simple, declarative interface for data collection and aggregation, inspired by selection and aggregation facilities in database query languages. Second, it intelligently distributes and executes aggregation queries in the sensor network in a time and power-efficient manner, and is sensitive to the resource constraints and lossy communication properties of wireless sensor networks. TAG processes aggregates *in the network* by computing over the data as it flows through the sensors, discarding irrelevant data and combining relevant readings into more compact records when possible.

TAG operates as follows: users pose aggregation queries from a powered, storage-rich basestation. Operators that implement the query are distributed into the network by piggybacking on the existing *ad hoc* networking protocol.

*This work has been partially supported by the NSF under grants IIS-0086057 and SI0122599, and by research funds from IBM, Intel, Microsoft, and the UC Micro program.

Sensors route data back towards the user through a routing tree rooted at the basestation. As data flows up this tree, it is aggregated according to an aggregation function and value-based partitioning specified in the query. As an example, consider a query that counts the number of nodes in a network of indeterminate size. First, the request to count is injected into the network. Then, each leaf node in the tree reports a count of 1 to their parent; interior nodes sum the count of their children, add 1 to it, and report that value to their parent. Counts propagate up the tree in this manner, and flow out at the root.

1.2 Overview of the Paper

The contributions of this paper are four-fold: first, we propose a simple, SQL-like declarative language for expressing aggregation queries over streaming sensor data and identify key properties of aggregation functions that affect the extent to which they can be efficiently processed inside the network. Second, we demonstrate how such in network execution can yield an order of magnitude reduction in communication compared to centralized approaches. Third, we show that by adopting a well-defined, declarative query language as a level of abstraction between the user and specific networking and routing protocols, a number of optimizations can be transparently applied to further reduce the data demands on the system. Finally, we show that our focus on a high-level language leads to useful end-to-end techniques for reducing the effects of network loss on aggregate results.

The remainder of the paper is structured as follows. In the next section, we briefly review the TinyOS hardware and software environment. Then, we discuss the syntax and semantics of queries in TAG and classify the types of aggregates supported by the system, focusing on the characteristics of aggregates that impact their performance and fault tolerance. We then present the core TAG algorithm and show how our solution satisfies the query requirements while providing performance and tolerance to network faults. We discuss several optimizations for improving the performance of the basic approach. Additionally, we include experimental results demonstrating the effectiveness and robustness of our algorithms in a simulation environment, as well as a brief study of a real-world deployment on TinyOS motes. Finally, we discuss related work and conclude.

2 Motes and Ad-Hoc Networks

In this section, we provide a brief overview of the mote hardware architecture, the TinyOS system, and an *ad hoc* routing algorithm for mote-based sensor networks.

2.1 Motes

Current generation TinyOS motes are equipped with a 4Mhz Atmel microprocessor with 4 kB of RAM and 128 kB of code space, a 917 MHz RFM radio running at 50 kb/s, and 512kB of EEPROM. An expansion slot accommodates a variety of sensor boards by exposing a number of analog input lines as well as popular chip-to-chip serial busses. Current sensor options include: light, temperature, magnetic field, acceleration, sound, and power.

The single-channel radio is half duplex, meaning motes cannot send and receive at the same time. Currently, the default TinyOS implementation uses a CSMA-like media access protocol with a random backoff scheme. Message delivery is unreliable by default, though applications can build up an acknowledgment layer. Often, a message acknowledgment can be obtained for free (see Section 2.2).

Power is supplied via an AA battery pack or a coin-cell attached through the expansion slot. The effective lifetime of the device is determined by this power supply. In turn, the power consumption of each sensor node tends to be dominated by the cost of transmitting and receiving messages. In terms of power consumption, transmitting a single bit of data is equivalent to 800 instructions. This energy tradeoff between communication and computation implies that many applications will benefit by processing the data inside the network rather than simply transmitting the sensor readings. An AA battery pack will allow a mote to send 5.52 million messages (if it does no other computation and only powers its radio up to transmit) which is equivalent to one message per second every day for about two months – not long if the goal is to deploy long lived, zero-maintenance ad-hoc sensor networks. Hence, power-conserving algorithms are particularly important. [1] As we will discuss in Section 4.1 , our design is amenable to very low power modes in which the radio is kept powered down for long periods of time.

To understand how data is routed in our ad-hoc aggregation network, two properties of radio communication need to be emphasized. First, radio is a broadcast medium so that any mote within hearing distance hears a message, irrespective of whether or not that mote is the intended recipient. Second, we only make use of symmetric links (where if mote a can hear mote b, b can also hear a.) As is common in ad-hoc protocols, asymmetric links are detected and blacklisted using a technique similar to that proposed in AODV [24].

Messages in the current generation of TinyOS are a fixed

[1] Note that as sensor devices become more integrated, it is expected that the ratio of communication to computation costs will get more important over time as silicon efficiency increases while the physical costs of pushing radio waves over the air remain constant.

size – by default, 30 bytes. Each device has a unique *sensor ID* that distinguishes it from others. All messages specify their recipient (or specify *broadcast*, meaning all available recipients), allowing motes to ignore messages not intended for them, although non-broadcast messages are received by all motes within range – unintended recipients simply drop messages not addressed to them.

2.2 Ad-Hoc Routing Algorithm

Given this overview of the mote environment, we now discuss how sensor devices route data. One common technique, which we sketch here, is to build a routing tree. We omit some details of this approach due to space constraints – a number routing protocols suitable for this purpose have been proposed; the reader is referred to [32, 13, 12, 14, 1] for more information. In general, TAG is agnostic to the choice of routing algorithm, requiring it to provide just two capabilities. First, it must be able to deliver query requests to all nodes in a network.[2] Second, it must be able to provide one or more routes from every node to the root of the network where aggregation data is being collected. These routes must guarantee that at most one copy of every message arrive (no duplicates are generated).

In the tree-based routing scheme, one mote is appointed to be the *root*, usually because it is the point where the user interfaces to the network. The root broadcasts a message asking motes to organize into a routing tree; in that message it specifies its own id and its *level*, or distance from the root (in this case, zero.) Any mote without an assigned level that hears this message assigns its own level to be the level in the message plus one. It also chooses the sender of the message as its *parent*, through which it will route messages to the root.

Each of these motes then rebroadcasts the routing message, inserting their own ids and levels. The routing message floods down the tree in this fashion, with each node rebroadcasting the message until all nodes have been assigned a level and a parent. These routing messages are periodically broadcast from the root, so that the process of topology discovery goes on continuously. This constant topology maintenance makes it relatively easy to adapt to network changes caused by mobility of certain nodes, or to the addition or deletion of motes. We describe a specific topology maintenance protocol used for our experiments on loss in Section 7.1 below. To maintain stability in the network, parents are retained unless a child does not hear from them for some long period of time, at which point it selects a new parent using this same process. We look in more detail at the robustness of this approach with respect to loss and its effect on aggregate values in Section 7.

[2]Note that, as an optimization, it may be useful for the routing layer to limit the extent to which queries are propagated based on properties of the *query* – for example, a short-lived query over constrained geographic area need not be sent to motes far away from that area. We reserve such optimizations for future work.

When a mote wishes to send a message to the root, it broadcasts a message addressed to its parent, which in turn forwards the message on to its parent, and so on, eventually reaching the root. In Section 4, we show how, as data is routed towards the root, it can be combined with data from other motes to efficiently combine routing and aggregation. Now, however, we turn to the syntax and semantics of aggregate queries in TAG.

3 Query Model and Environment

Given our goal of allowing users to pose declarative queries over sensor networks, we needed a language for expressing such queries. Rather than inventing our own, we chose to adopt a SQL-style query syntax. We support SQL-style queries (without joins) over a single table called `sensors`, whose schema is known at the base station. As is the case in Cougar [23], this table can be thought of as an append-only relational table with one attribute per input of the motes (e.g., temperature, light.) In TAG, we focus on the problem of aggregate sensor readings, though facilities for collecting individual sensor readings also exist.

Before describing the semantics of queries in general, we begin with an example query. Consider a user who wishes to monitor the occupancy of the conference rooms on a particular floor of a building, which she chooses to do by using microphone sensors attached to motes, and looking for rooms where the average volume is over some threshold (assuming that rooms can have multiple sensors). Her query could be expressed as:

```
SELECT AVG(volume),room FROM sensors
  WHERE floor = 6
  GROUP BY room
  HAVING AVG(volume) > threshold
  EPOCH DURATION 30s
```

This query partitions motes on the 6th floor according to the room in which they are located (which may be a hard-coded constant in each device, or may be determined via some localization component available to the devices.) The query then reports all rooms where the average volume is over a specified threshold. Updates are delivered every 30 seconds, although the user may deregister her query at any time.

In general, queries in TAG have the form:

```
SELECT {agg(expr), attrs} FROM sensors
  WHERE {selPreds}
  GROUP BY {attrs}
  HAVING {havingPreds}
  EPOCH DURATION i
```

With the exception of the `EPOCH DURATION` clause, the semantics of this statement are similar to SQL aggregate queries. The SELECT clause specifies an arbitrary arithmetic expression over one or more aggregation attributes. We expect that the common case here is that *expr* will simply be the name of a single attribute. Attrs (optionally) selects the attributes by which the sensor readings are partitioned ; these can be any subset of attrs that appear in the GROUP BY clause. The syntax of the *agg* clause is discussed below; note that multiple aggreggates may be computed in a single query. The WHERE clause filters out individual sensor readings before they are aggregated. Such predicates can typically be executed locally at the mote before readings are communicated, as in [23, 18]. The GROUP BY clause specifies an attribute based partitioning of sensor readings. Logically, each reading belongs to exactly one group, and the evaluation of the query is a table of group identifiers and aggregate values. The HAVING clause filters that table by suppressing groups that do not satisfy the havingPreds predicates.

The primary semantic difference between TAG queries and SQL queries is that the output of a TAG query is a stream of values, rather than a single aggregate value (or batched result). In monitoring applications, such continuous results are often more useful than a single, isolated aggregate, as they allow users to understand how the network is behaving over time and observe transient effects (such as message losses) that make individual results, taken in isolation, hard to interpret. In these *stream semantics*, each record consists of one *<group id,aggregate value>* pair per group. Each group is time-stamped and the readings used to compute an aggregate record all belong to the same time interval, or *epoch*. The duration of each epoch is the argument of the EPOCH DURATION clause, which specifies the amount of time (in seconds) devices wait before acquiring and transmitting each successive sample. This value may be as large as the user desires; it must be at least as long as the time it takes for a mote to process and transmit a single radio message and do some local processing – about 30 ms (including average MAC backoff in a low-contention environment) for current generation motes (yielding a maximum sample rate of about 33 samples per second.) In section 4.1, we discuss situations that require longer lower bounds on epoch duration.

3.1 Structure of Aggregates

The problem of computing aggregate queries in large clusters of nodes has been addressed in the context of shared-nothing parallel query processing environments [26]. Like sensor networks, those environments require the coordination of a large number of nodes to process aggregations. Thus, while the severe bandwidth limitations, lossy communications, and variable topology of sensor networks mean that the specific implementation techniques used in the two environments must differ, it is still useful to leverage the techniques for aggregate decomposition used in database systems [2, 35].

The approach used in such systems (and followed in TAG) is to implement agg via three functions: a merging function f, an initializer i, and an evaluator, e.

In general, f has the following structure:

$$< z >= f(< x >, < y >)$$

where $< x >$ and $< y >$ are multi-valued *partial state records*, computed over one or more sensor values, representing the intermediate state over those values that will be required to compute an aggregate. $< z >$ is the partial-state record resulting from the application of function f to $< x >$ and $< y >$. For example, if f is the merging function for AVERAGE, each partial state record will consist of a pair of values: SUM and COUNT, and f is specified as follows, given two state records $< S_1, C_1 >$ and $< S_2, C_2 >$:

$$f(< S_1, C_1 >, < S_2, C_2 >) = < S_1 + S_2, C_1 + C_2 >$$

The initializer i is needed to specify how to instantiate a state record for a single sensor value; for an AVERAGE over a sensor value of x, the initializer $i(x)$ returns the tuple $< x, 1 >$. Finally, the evaluator e takes a partial state record and computes the actual value of the aggregate. For AVERAGE, the evaluator $e(< S, C >)$ simply returns S/C.

These three functions can easily be derived for the basic SQL aggregates; in general, any operation that can be expressed as commutative applications of a binary function is expressible.

3.2 Taxonomy of Aggregates

Given our basic syntax and structure of aggregates, an obvious question remains: what aggregate functions can be expressed in TAG? The original SQL specification offers just five options: COUNT, MIN, MAX, SUM, and AVERAGE. Although these basic functions are suitable for a wide range of database applications, we did not wish to constrain TAG to only these choices. For this reason, we present a general classification of aggregate functions and show how the dimensions of that classification affect the performance of TAG throughout the paper. We will assume that when aggregation functions are registered with TAG, they are classified along the dimensions described below.[3]

[3]We omit a detailed discussion of how new aggregate functions are registered with motes. For now, aggregates are pre-compiled into motes. Virtual-machine languages recently proposed for TinyOS-style [16] motes could also be used for this purpose.

	MAX, MIN	COUNT, SUM	AVERAGE	MEDIAN	COUNT DISTINCT[4]	HISTOGRAM[5]	Section
Duplicate Sensitive	No	Yes	Yes	Yes	No	Yes	Section 7.5
Exemplary (E), Summary (S)	E	S	S	E	S	S	Section 6.2
Monotonic	Yes	Yes	No	No	Yes	No	Section 4.2
Partial State	Distributive	Distributive	Algebraic	Holistic	Unique	Content-Sensitive	Section 5.1

Table 1: Classes of aggregates

We classify aggregates according to four properties that are particularly important to sensor networks. Table 1 shows how specific aggregation functions can be classified according to these properties, and indicates the sections of the paper where the various dimensions of the classification are emphasized.

The first dimension is duplicate sensitivity. *Duplicate insensitive* aggregates are unaffected by duplicate readings from a single device while *duplicate sensitive* aggregates will change when a duplicate reading is reported. Duplicate sensitivity implies restrictions on network properties and on certain optimizations, as described in Section 7.5.

Second, *exemplary* aggregates return one or more representative values from the set of all values; *summary* aggregates compute some property over all values. This distinction is important because exemplary aggregates behave unpredictably in the face of loss, and, for the same reason, are not amenable to sampling. Conversely, for summary aggregates, the aggregate applied to a subset can be treated as a robust approximation of the true aggregate value, assuming that either the subset is chosen randomly, or that the correlations in the subset can be accounted for in the approximation logic.

Third, *monotonic* aggregates have the property that when two partial state records, s_1 and s_2, are combined via f, the resulting state record s' will have the property that either $\forall s_1, s_2, e(s') \geq MAX(e(s_1), e(s_2))$ or $\forall s_1, s_2, e(s') \leq MIN(e(s_1), e(s_2))$. This is important when determining whether some predicates (such as HAVING) can be applied *in network*, before the final value of the aggregate is known. Early predicate evaluation saves messages by reducing the distance that partial state records must flow up the aggregation tree.

The fourth dimension relates to the amount of state required for each partial state record. For example, a partial AVERAGE record consists of a pair of values, while a partial COUNT record constitutes only a single value. Though TAG correctly computes any aggregate that conforms to the specification of f in Section 3 above, its performance is inversely related to the amount of intermediate state required per aggregate. The first three categories of this dimension (e.g. distributive, algebraic, holistic) were initially presented in work on data-cubes [9].

- In *Distributive* aggregates, the partial state is simply the aggregate for the partition of data over which they are computed. Hence the size of the partial state records is the same as the size of the final aggregate.
- In *Algebraic* aggregates, the partial state records are not themselves aggregates for the partitions, but are of constant size.
- In *Holistic* aggregates, the partial state records are proportional in size to the set of data in the partition. In essence, for holistic aggregates no useful partial aggregation can be done, and all the data must be brought together to be aggregated by the evaluator.
- *Unique* aggregates are similar to holistic aggregates, except that the amount of state that must be propagated is proportional to the number of distinct values in the partition.
- In *Content-Sensitive* aggregates, the partial state records are proportional in size to some (perhaps statistical) property of the data values in the partition. Many approximate aggregates proposed recently in the database literature are content-sensitive. Examples of such aggregates include fixed-width histograms, wavelets, and so on; see [3] for an overview of such functions.

In summary, we have classified aggregates according to their state requirements, tolerance of loss, duplicate sensitivity, and monotonicity. We will refer back to this classification throughout the text, as these properties will determine the applicability of communication optimizations we present later. Understanding how aggregates fit into these categories is a cross-cutting issue that is critical (and useful) in many aspects of sensor data collection.

Note that our formulation of aggregate functions, combined with this taxonomy, is flexible enough to encompass a wide range of sophisticated operations. For example, we have implemented (in the simulator described in Section 5 below), an *isobar finding* aggregate. This is a duplicate-insensitive, summary, monotonic, content-sensitive aggregate that builds a topological map representing discrete bands of one attribute (light, for example) plotted against two other attributes (x and y position in some local coordinate space, for example.)

[4]The HISTOGRAM aggregate sorts sensor readings into fixed-width buckets and returns the size of each bucket; it is content-sensitive because the number of buckets varies depending on how widely spaced sensor readings are.

[5]COUNT DISTINCT returns the number of distinct values reported across all motes.

3.3 Attribute Catalog

Queries in TAG contain named attributes. Some mechanism is needed to allow users to determine the set of attributes they may query, and to allow motes to advertise the attributes they can provide. In TAG, we include on each mote a small *catalog* of attributes. This catalog can be searched for attributes of a specific name, or iterated through. To limit the burden of reporting catalog information from motes, we assume the central query processor caches or stores the attributes of all motes it may access.

When a TAG sensor receives a query, it converts named fields into local catalog identifiers. Nodes lacking attributes specified in the query simply tag missing attributes as NULL in their result records (alternatively, the query could specify that the lacking node should opt out of the query.) This technique increases the scalability of large sensor network deployments as it does not require all nodes to have global knowledge of all attributes.

As in relational databases, partial state records resulting from the evaluation of a query have the same layout across all nodes. Thus, tuples in TAG need not be self-describing; attribute names are not carried with results, leading to a significant reduction in the amount of data that must be propagated with each tuple. At the same time, it is not necessary for all nodes to have identical catalogs, which allows heterogeneous sensing capabilities and incremental deployment of motes.

Attributes in TAG may be direct representations of sensor values, such as light or temperature, or may be introspective, such as remaining energy or network neighborhood information. More generally, they can represent time-varying statistics over local sensor values, such as an exponentially decaying average of the last n light readings, or more complicated attributes such as a room number, GPS coordinate, or relative distance to some neighbor from a localization component. Individual software components in TinyOS choose which attributes they will make available, and provide an accessor function for acquiring the next attribute reading.

4 In Network Aggregates

Given the simple routing protocol from Section 2.2 and our query model, we now discuss the implementation of the core TAG algorithm for in network aggregation.

A naive implementation of sensor network aggregation would be to use a centralized, *server-based* approach where all sensor readings are sent to the base station, which then computes the aggregates. In TAG, however, we compute aggregates in network whenever possible, because, if properly implemented, this approach can be lower in number of message transmissions, latency, and power consumption than the server-based approach. We will measure the advantage of in network aggregation in Section 5 below; first, we present the basic algorithm in detail. We first consider the operation of the basic approach in the absence of grouping; we show how to extend it with grouping in Section 4.2.

4.1 Tiny Aggregation

TAG consists of two phases: a *distribution* phase, in which aggregate queries are pushed down into the network, and a *collection* phase, where the aggregate values are continually routed up from children to parents. Recall that our query semantics partition time into epochs of duration i, and that we must produce a single aggregate value (when not grouping) that combines the readings of all devices in the network during that epoch.

Given our goal of using as few messages as possible, the collection phase must ensure that parents in the routing tree wait until they have heard from their children before propagating an aggregate value for the current epoch. We will accomplish this by having parents subdivide the epoch such that children are required to deliver their partial state records during a parent-specified time interval. This interval is selected such that there is enough time for the parent to combine partial state records and propagate its own record to its parent.

When a mote p receives a request to aggregate, r, either from another mote or from the user, it awakens, synchronizes its clock according to timing information in the message, and prepares to participate in aggregation. In the tree based routing scheme, p chooses the sender of the message as its parent. In addition to the information in the query, r includes the interval when the sender is expecting to hear partial state records from p. p then forwards the query request r down the network, setting this delivery interval for children to be slightly before the time its parent expects to see p's partial state record. In the tree-based approach, this forwarding consists of a broadcast of r, to include any nodes that did not hear the previous round, and include them as children (if it has any.) These nodes continue to forward the request in this manner, until the query has been propagated throughout the network.

During the epoch after query propagation, each mote listens for messages from its children during the interval it specified when forwarding the query. It then computes a partial state record consisting of the combination of any child values it heard with its own local sensor readings. Finally, during the transmission interval requested by its parent, the mote transmits this partial state record up the network. Figure 1 illustrates the process. Notice that parents listen for longer than the transmission interval they

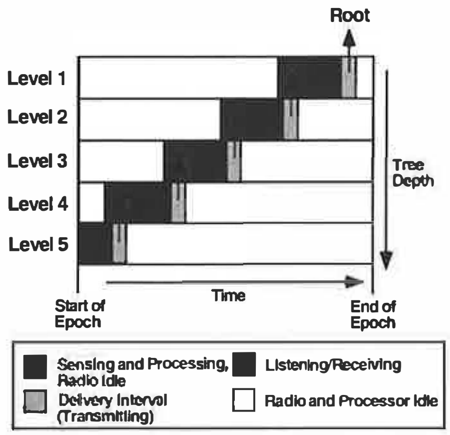


Figure 1: *Partial state records flowing up the tree during an epoch.*

specified, to overcome limitations in the quality of clock synchronization algorithms between parents and children. In this way, aggregates flow back up the tree interval-by-interval. Eventually, a complete aggregate arrives at the root. During each subsequent epoch, a new aggregate is produced. Notice that, for a significant portion of each epoch, motes are idle and can enter a low power state.

This scheme begs the question of how parents choose the duration of the interval in which they will receive values. It needs to be long enough such that all of a node's children can report, but not so long that the epoch ends before nodes deep in the tree can schedule their communication. Furthermore, longer intervals require radios to be powered up for more time, which consumes precious energy. In general, the proper choice of duration for these intervals is somewhat environment specific, as it depends on the density of radio cells and "bushiness" of the network topology. For the purposes of the simulations and experiments in this paper, we assume the network has a maximum depth d, and set the duration of each interval to be (EPOCH DURATION)/d, with nodes at level i transmitting during the ith interval. We rely on the TinyOS MAC layer [32] to avoid collisions between nodes transmitting during the same interval. Note that this provides a lower-bound on the EPOCH DURATION and constrains the maximum sample rate of the network, since the epoch must be long enough for partial state records from the bottom of the tree to propagate to the root.

To increase the sample rate, one could consider pipelining the communications schedule shown in Figure 1. With pipelining, the output of the network would be delayed by one or more epochs, as some nodes would wait until the next epoch to report the aggregates they collected during the current epoch. In exchange for such delays, the effective sample rate of the system is increased (for the same reason that pipelining a long processor stage increases the clock rate of a CPU.) We do not consider such schemes in detail here; we discussed a fully-pipelined approach to aggregation in a workshop submission [20].

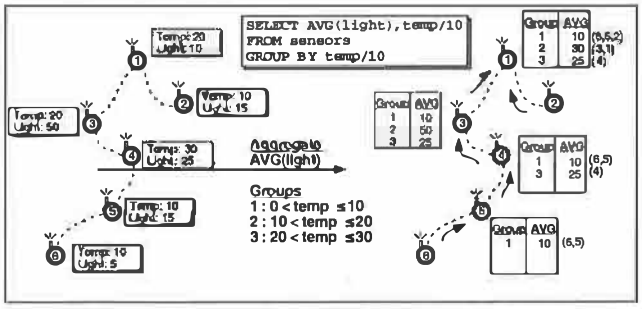
In Section 5.1 we show how TAG can provide an order of magnitude decrease in communications costs over a centralized approach. However, before discussing performance, we extend the approach to support grouping.

4.2 Grouping

Grouping in TAG is functionally equivalent to the GROUP BY clause in SQL: each sensor reading is placed into exactly one group, and groups are partitioned according to an expression over one or more attributes. The basic grouping technique is to push the expression down with the query, ask nodes to choose the group they belong to, and then, as answers flow back, update aggregate values in the appropriate groups.

Partial state records are aggregated just as in the approach described above, except that those records are now tagged with a group id. When a node is a leaf, it applies the grouping expression to compute a group id. It then tags its partial state record with the group and forwards it on to its parent. When a node receives an aggregate from a child, it checks the group id. If the child is in the same group as the node, it combines the two values using the combining function f. If it is in a different group, it stores the value of the child's group along with its own value for forwarding in the next epoch. If another child message arrives with a value in either group, the node updates the appropriate aggregate. During the next epoch, the node sends the value of all the groups about which it collected information during the previous epoch, combining information about multiple groups into a single message as long as message size permits. Figure 2 shows an example of computing a query grouped by temperature that selects average light readings.

Recall that queries may contain a HAVING clause, which constrains the set of groups in the final query result. This predicate can sometimes be passed into the network along



```
SELECT AVG(light),temp/10
FROM sensors
GROUP BY temp/10
```

Group	AVG	
1	10	(6,5,2)
2	30	(3,1)
3	25	(4)

Group	AVG
1	10
2	50
3	25

Group	AVG	
1	10	(6,5)
3	25	(4)

Group	AVG	
1	10	(6,5)

Figure 2: *A sensor network (left) with an in network, grouped aggregate applied to it (right).* Parenthesized numbers represent nodes that contribute to the average

with the grouping expression. The predicate is only sent if it can potentially be used to reduce the number of messages that must be sent: for example, if the predicate is of the form `MAX(attr) < x`, then information about groups with `MAX(attr) ≥ x` need not be transmitted up the tree, and so the predicate is sent down into the network. When a node detects that a group does not satisfy a `HAVING` clause, it can notify other nodes in the network of this information to suppress transmission and storage of values from that group. Note that `HAVING` clauses can be pushed down only for monotonic aggregates; non-monotonic aggregates are not amenable to this technique. However, not all `HAVING` predicates on monotonic aggregates can be pushed down; for example, `MAX(attr) > x` cannot be applied in the network because a node cannot know that, just because its local value of *attr* is less than x, the `MAX` over the entire group is less than x.

Grouping introduces an additional problem: the number of groups can exceed available storage on any one (non-leaf) device. Our proposed solution is to evict one or more groups from local storage. Once an eviction victim is selected, it is forwarded to the node's parent, which may choose to hold on to the group or continue to forward it up the tree. Notice that a single node may evict several groups in a single epoch (or the same group multiple times, if a bad victim is selected). This is because, once group storage is full, if only one group is evicted at a time, a new eviction decision must be made every time a value representing an unknown or previously evicted group arrives. Because groups can be evicted, the base station at the top of the network may be called upon to combine partial groups to form an accurate aggregate value. Evicting partially computed groups is known as *partial preaggregation*, as described in [15].

Thus, we have shown how to partition sensor readings into a number of groups and properly compute aggregates over those groups, even when the amount of group information exceeds available storage in any one device. We will briefly mention experiments with grouping and group eviction policies in Section 5.2. First, we summarize some of the additional benefits of TAG.

4.3 Additional Advantages of TAG

The principal advantage of TAG is its ability to dramatically decrease the communication required to compute an aggregate versus a centralized aggregation approach. However, TAG has a number of additional benefits.

One of these is its ability to tolerate disconnections and loss. In sensor environments, it is very likely that some aggregation requests or partial state records will be garbled, or that devices will move or run out of power. These losses will invariably result in some nodes becoming *lost*, either without a parent or not incorporated into the aggregation network during the initial flooding phase. If we include information about queries in partial state records, lost nodes can reconnect by listening to other node's state records – not necessarily intended for them – as they flow up the tree. We revisit the issue of loss in Section 7.

A second advantage of the TAG approach is that, in most cases, each mote is required to transmit only a single message per epoch, regardless of its depth in the routing tree. In the centralized (non TAG) case, as data converges towards the root, nodes at the top of the tree are required to transmit significantly more data than nodes at the leaves; their batteries are drained faster and the lifetime of the network is limited. Furthermore, because the top of the routing tree must forward messages for every node in the network, the maximum sample rate of the system is inversely proportional to the total number of nodes. To see this, consider a radio channel with a capacity of n messages per second. If m motes are participating in a centralized aggregate, to obtain a sample rate of k samples per second, $m \times k$ messages must flow through the root during each epoch. $m \times k$ must be no larger than n, so the sample rate k can be at most n/m messages per mote per epoch, regardless of the network density. When using TAG, the maximum transmission rate is limited instead by the occupancy of the largest radio-cell; in general, we expect that each cell will contain far fewer than m motes.

Yet another advantage of TAG is that, by explicitly dividing time into epochs, a convenient mechanism for idling the processor is obtained. The long idle times in Figure 1 show how this is possible; during these intervals, the radio and processor can be put into deep sleep modes that use very little power. Of course, some bootstrapping phase is needed where motes can learn about queries currently in the system, acquire a parent, and synchronize clocks; a simple strategy involves requiring that every node wake up infrequently but periodically to advertise this information and that devices that have not received advertisements from their neighbors listen for several times this period between sleep intervals. Research on energy aware MAC protocols [34] presents a similar scheme in detail. That work also discusses issues such as time synchronization resolution and the maximum sleep duration to avoid the adverse effects of clock skew on individual devices.

Taken as a whole, these TAG features provide users with a stream of aggregate values that changes as sensor readings and the underlying network change. These readings are provided in an energy and bandwidth efficient manner.

5 Simulation-Based Evaluation

In this section, we present a simulation environment for TAG and evaluate its behavior using this simulator. We also have an initial, real-world deployment; we discuss its performance at the end of the paper, in Section 8.

To study the algorithms presented in this paper, we simulated TAG in Java. The simulator models mote behavior at a coarse level: time is divided into units of epochs, messages are encapsulated into Java objects that are passed directly into nodes without any model of the time to send or decode. Nodes are allowed to compute or transmit arbitrarily within a single epoch, and each node executes serially. Messages sent by all nodes during one epoch are delivered in random order during the next epoch to model a parallel execution. Note that this simulator cannot account for certain low-level properties of the network: for example, because there is no fine-grained model of time, it is not possible to model radio contention at a byte level.

Our simulation includes an interchangeable communication model that defines connectivity based on geographic distance. Figure 3 shows screenshots of a visualization component of our simulation; each square represents a single device, and shading (in these images) represents the number of radio hops the device is from the root (center); darker is closer. We measure the size of networks in terms of *diameter*, or width of the sensor grid (in nodes). Thus, a diameter 50 network contains 2500 devices.

We have run experiments with three communications models; 1) a *simple* model, where nodes have perfect (lossless) communication with their immediate neighbors, which are regularly placed (Figure 3(a)), 2) a *random* placement model (Figure 3(b)), and 3) a *realistic* model that attempts to capture the actual behavior of the radio and link layer on TinyOS motes (Figure 3(c).) In this last model, notice that the number of hops from a particular node to the root is no longer directly proportional to the geographic distance between the node and the root, although the two values are still related. This model uses results from real world experiments [7] to approximate the actual loss characteristics of the TinyOS radio. Loss rates are high in in the realistic model: a pair of adjacent nodes loses more than 20% of the traffic between them. Devices separated by larger distances lose still more traffic.

The simulator also models the costs of topology maintenance: if a node does not transmit a reading for several epochs (which will be the case in some of our optimizations below), that node must periodically send a *heartbeat* to advertise that it is still alive, so that its parents and children know to keep routing data through it. The interval between heartbeats can be chosen arbitrarily; choosing a longer interval means fewer messages must be sent, but requires nodes to wait longer before deciding that a parent or child has disconnected, making the network less adaptable to rapid change.

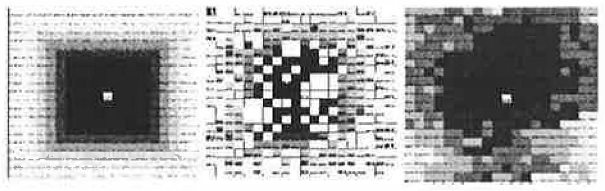

(a) Simple (b) Random (c) Realistic

Figure 3: *The TAG Simulator, with Three Different Communications Models, Diameter = 20.*

This simulation allows us to measure the the number of bytes, messages, and partial state records sent over the radio by each mote. Since we do not simulate the mote CPU, it does not give us an accurate measurement of the number of instructions executed in each mote. It does, however, allow us to obtain an approximate measure of the state required for various algorithms, based on the size of the data structures allocated by each mote.

Unless otherwise specified, our experiments are over the simple radio topology in which there is no loss. We also assume sensor values do not change over the course of a single simulation run.

5.1 Performance of TAG

In the first set of experiments, we compare the performance of the TAG in network approach to centralized approaches on queries for the different classes of aggregates discussed in Section 3.2. Centralized aggregates have the same communications cost irrespective of the aggregate function, since all data must be routed to the root. For this experiment, we compared this cost to the number of bytes required for distributive aggregates (`MAX` and `COUNT`), an algebraic aggregate (`AVERAGE`), a holistic aggregate (`MEDIAN`), a content-sensitive aggregate (`HISTOGRAM`), and a unique aggregate (`COUNT DISTINCT`); the results are shown in Figure 4.

Values in this experiment represent the steady-state cost to extract an additional aggregate from the network once the query has been propagated; the cost to flood a request down the tree in not considered.

In our 2500 node ($d = 50$) network, `MAX` and `COUNT` have the same cost when processed in the network, about 5000 bytes per epoch (total over all nodes), since they both send just a single integer per partial state record; similarly `AVERAGE` requires just two integers, and thus always has double the cost of the distributive aggregates. `MEDIAN` costs the same as a centralized aggregate, about 90000 bytes per epoch, which is significantly more expensive

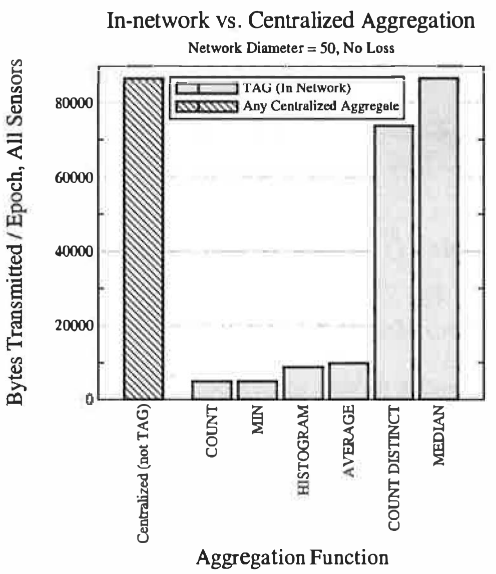


Figure 4: *In network Vs. Centralized Aggregates*

than other aggregates, especially for larger networks, as parents have to forward all of their children's values to the root. COUNT DISTINCT is only slightly less expensive (73000 bytes), as there are few duplicate sensor values; a less uniform sensor-value distribution would reduce the cost of this aggregate. For the HISTOGRAM aggregate, we set the size of the fixed-width buckets to be 10; sensor values ranged over the interval [0..1000]. At about 9000 messages per epoch, HISTOGRAM provides an efficient means for extracting a density summary of readings from the network.

Note that the benefit of TAG will be more or less pronounced depending on the topology. In a flat, single-hop environment, where all motes are directly connected to the root, TAG is no better than the centralized approach. For a topology where n motes are arranged in a line, centralized aggregates will require $n^2/2$ partial state records to be transmitted, whereas TAG will require only n records.

Thus, we have shown that, for our simulation topology, in network aggregation can reduce communication costs by an order of magnitude over centralized approaches, and that, even in the worst case (such as with MEDIAN), it provides performance equal to the centralized approach.

5.2 Grouping Experiments

We also ran several experiments to measure the performance of grouping in TAG, focusing on the behavior of various eviction techniques. We tried a number of simple eviction policies, but found that the choice of policy made little difference for any of the sensor-value distributions we tested – in the most extreme case, the difference between the best and worst case eviction policy accounted for less than 10% of the total messages. Due to the relative insignificance of these results and space limitations, we omit a detailed discussion of the merits of various eviction policies.

6 Optimizations

In this section, we present several techniques to improve the performance and accuracy of the basic approach described above. Some of these techniques are function dependent; that is, they can only be used for certain classes of aggregates. Also note that, in general, these techniques can be applied in a user-transparent fashion, since they are not explicitly a part of the query syntax and do not affect the semantics of the results.

6.1 Taking Advantage of A Shared Channel

In our discussion of aggregation algorithms up to this point, we have largely ignored the fact that motes communicate over a shared radio channel. The fact that every message is effectively broadcast to all other nodes within range enables a number of optimizations that can significantly reduce the number of messages transmitted and increase the accuracy of aggregates in the face of transmission failures.

In Section 4.3, we saw an example of how a shared channel can be used to increase message efficiency when a node misses an initial request to begin aggregation: it can initiate aggregation even after missing the start request by *snooping* on the network traffic of nearby nodes. When it hears another device reporting an aggregate, it can assume it too should be aggregating. By allowing nodes to examine messages not directly addressed to them, motes are automatically integrated into the aggregation. Note that snooping does not require nodes to listen all the time; by listening at predefined intervals (which can be short once a mote has time-synchronized with its neighbors), duty cycles can be kept quite low.

Snooping can also be used to reduce the number of messages sent for some classes of aggregates. Consider computing a MAX over a group of motes: if a node hears a peer reporting a maximum value greater than its local maximum, it can elect to not send its own value and be sure it will not affecting the value of the final aggregate.

6.2 Hypothesis Testing

The snooping example above showed that we only need to hear from a particular node if that node's value will affect the end value of the aggregate. For some aggregates, this fact can be exploited to significantly reduce the number of nodes that need to report. This technique can be generalized to an approach we call *hypothesis testing*. For certain classes of aggregates, if a node is presented with a guess as to the proper value of an aggregate, it can decide locally whether contributing its reading and the readings of its children will affect the value of the aggregate.

For MAX, MIN and other monotonic, exemplary aggregates, this technique is directly applicable. There are a number of ways it can be applied – the snooping approach, where nodes suppress their local aggregates if they hear other aggregates that invalidate their own, is one. Alternatively, the root of the network (or any subtree of the network) seeking an exemplary sensor value, such as a MIN, might compute the minimum sensor value m over the highest levels of the subtree, and then abort the aggregate and issue a new request asking for values less than m over the whole tree. In this approach, leaf nodes need not send a message if their value is greater than the minimum observed over the top k levels; intermediate nodes, however, must still forward partial state records, so even if their value is suppressed, they may still have to transmit.

Assuming for a moment that sensor values are independent and uniformly distributed, then a particular leaf node must transmit with probability $1/b^k$ (where b is the branching factor, so $1/b^k$ is the number of nodes in the top k levels), which is quite low for even small values of k. For bushy routing trees, this technique offers a significant reduction in message transmissions – a completely balanced routing tree would cut the number of messages required to $1/k$. Of course, the performance benefit may not be as substantial for other, non-uniform, sensor value distributions; for instance, a distribution in which all sensor readings are clustered around the minimum will not allow many messages to be saved by hypothesis testing. Similarly, less balanced topologies (e.g. a line of nodes) will not benefit from this approach.

For summary aggregates, such as AVERAGE or VARIANCE, hypothesis testing via a guess from the root can be applied, although the message savings are not as dramatic as with monotonic aggregates. Note that the snooping approach cannot be used: it only applies to monotonic, exemplary aggregates where values can be suppressed locally without any information from a central coordinator. To obtain any benefit with summary aggregates and hypothesis testing, the user must define a fixed-size error bound that he or she is willing to tolerate over the value of the aggregate; this error is sent into the network along with the hypothesis value.

Consider the case of an AVERAGE: any device whose sensor value is within the error bound of the hypothesis value need not answer – its parent will then assume its value is the same as the approximate answer and count it accordingly (to apply this technique with AVERAGE, parents must know how many children they have.) It can be shown that the total computed average will not be off from

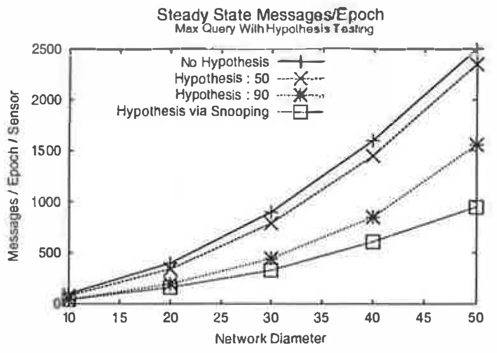


Figure 5: *Benefit of Hypothesis Testing for* MAX

the actual average by more than the error bound, and leaf nodes with values close to the average will not be required to report. Obviously, the value of this scheme depends on the distribution of sensor values. In a uniform distribution, the fraction of leaves that need not report approximates the size of the error bound divided by the size of the sensor value distribution interval. If values are normally distributed, a much larger fraction of leaves do not report.

We conducted a simple experiment to measure the benefit of hypothesis testing and snooping for a MAX aggregate. The results are shown in Figure 5. In this experiment, sensor values were uniformly distributed over the range [0..100], and a hypothesis was made at the root. Notice that the performance savings are nearly two-fold for a hypothesis of 90. We compared the hypothesis testing approach with the snooping approach (which will be effective even in a non-uniform distribution); surprisingly, snooping beat the other approaches by offering a nearly three-fold performance increase over the no-hypothesis case. This is because in the densely packed simple node distribution, most devices have three or more neighbors to snoop on, suggesting that only about one in four devices will have to transmit. With topology maintenance and forwarding of child values by parents, the savings by snooping is reduced to a factor of three.

7 Improving Tolerance to Loss

Up to this point in our experiments we used a reliable environment where no messages were dropped and no nodes disconnected or went offline. In this section, we address the problem of loss and its effect on the algorithms presented thus far. Unfortunately, unlike in traditional database systems, communication loss is a a fact of life in the sensor domain; the techniques described in the section seek to mitigate that loss.

7.1 Topology Maintenance and Recovery

TAG is designed to sit on top of a shifting network topology that adapts to the appearance and disappearance of nodes. Although a study of mechanisms for adapting topology is not central to this paper, for completeness we describe a basic topology maintenance and recovery algorithm which we use in both our simulation and implementation. This approach is similar to techniques used in practice in existing TinyOS sensor networks, and is derived from the general techniques proposed in the ad-hoc networking literature[28, 22].

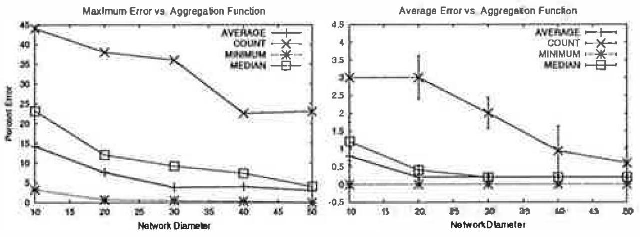
Networking faults are monitored and adapted to at two levels: First, each node maintains a small, fixed sized list of neighbors, and monitors the quality of the link to each of those neighbors by tracking the proportion of packets received from each neighbor. This is done via a locally unique sequence number associated with each message by its sender. When a node n observes that the link quality to its parent p is significantly worse than that of some other node p', it chooses p' as its new parent *if* p' is as close or closer to the root as p and p' does not believe n is its parent (the latter two conditions prevent routing cycles.)

Second, when a node observes that it has not heard from its parent for some fixed period of time (relative to the epoch duration of the query it is currently running), it assumes its parent has failed or moved away. It resets its local level (to ∞) and picks a new parent from the neighbor table according to the quality metric used for link-quality. Note that this can cause a parent to select a node in the routing subtree underneath it as its parent, so child nodes must reselect their parent (as though a failure had occurred) when they observe that their own parent's level has gone up.

Note that switching parents does not introduce the possibility of multiple records arriving from a single node, as each node transmits only once per epoch (even if it switches parents during that epoch.) Parent switching can cause temporary disconnections (and thus additional lost records) in the topology, however, due to children selecting a new parent when their parent's level goes up.

7.2 Effects of A Single Loss

We first study the effect that a single device going offline has on the value of the aggregate; this is an important measurement because it gives some intuition about the magnitude of error that a single loss can generate. Note that, because we are doing hierarchical aggregation, a single mote going offline causes the entire subtree rooted at the node to be (at least temporarily) disconnected. In this first experiment we used the simple topology, with sensor readings chosen from the uniform distribution over [1..1000]. After running the simulation for several epochs, we selected, uniformly and at a random, a node to disable. In this environment, children of the disabled node were temporarily disconnected but eventually their values were reintegrated into the aggregate once they rediscovered their parents. Note that the amount of time taken for lost nodes to reintegrate is directly proportional to the depth of the lost node, so we did not measure it experimentally. Instead, we measured the maximum temporary deviation from the true value of the aggregate that the loss caused in the perceived aggregate value at the root during any epoch. This maximum was computed by performing 100 runs at each data point and selecting the largest error reported in any run. We also report the average of the maximum error across all 100 runs.



(a) Maximum Error (b) Average Error

Figure 6: *Effect of a Single Loss on Various Aggregate Functions.* Computed over a total of 100 runs at each point. Error bars indicate standard error of the mean, 95% confidence intervals.

Figure 6 shows the results of this experiment. Note that the maximum loss (Figure 6(a)) is highly variable and that some aggregates are considerably more sensitive to loss than others. COUNT, for instance, has a very large error in the worst case: if a node that connects the root to a large portion of the network is lost, the temporary error will be very high. The variability in maximum error is because a well connected subtree is not always selected as the victim. Indeed, assuming some uniformity of placement (e.g. the devices are not arranged in a line), as the network size increases, the chances of selecting such a node go down, since a larger proportion of the nodes are towards the leaves of the tree. In the average case(Figure 6(b)), the error associated with a COUNT is not as high: most losses do not result in a large number of disconnections. Note that MIN is insensitive to loss in this uniform distribution, since several nodes are at or near the true minimum. The error for MEDIAN and AVERAGE is less than COUNT and more than MIN: both are sensitive to the variations in the number of nodes, but not as dramatically as COUNT.

7.3 Effect of Realistic Communication

In the second experiment, we examine how well TAG performs in the realistic simulation environment (discussed in Section 5 above). In such an environment, without some technique to counteract loss, a large number of partial state records will invariably be dropped and not reach the root of the tree. We ran an experiment to measure the effect of this loss in the realistic environment. The simulation ran until the first aggregate arrived at the root, and then the average number of motes involved in the aggregate over the next several epochs was measured. The "No Cache" line of Figure 7 shows the performance of this approach; at diameter 10, about 40% of the partial state records are reflected in the aggregate at the root; by diameter 50, this percentage has fallen to less than 10%. Performance falls off as the number of hops between the average node and the root increases, since the probability of loss is compounded by each additional hop. Thus, the basic TAG approach presented so far, running on current prototype hardware (with its very high loss rates), is not highly tolerant to loss, especially for large networks. Note that any centralized approach would suffer from the same loss problems.

7.4 Child Cache

To improve the quality of aggregates, we propose a simple caching scheme: parents remember the partial state records their children reported for some number of rounds, and use those previous values when new values are unavailable due to lost child messages. As long as the duration of this memory is shorter than the interval at which children select new parents, this technique will increase the number of nodes included in the aggregate without over-counting any nodes. Of course, caching tends to temporally smear the aggregate values that are computed, reducing the temporal resolution of individual readings and possibly making caching undesirable for some workloads. Note that caching is a simple form of interpolation where the interpolated value is the same as the previous value. More sophisticated interpolation schemes, such as curve fitting or statistical observations based on past behavior, could be also be used.

We conducted some experiments to show the improvement caching offers over the basic approach; we allocate a fixed size buffer at each node and measure the average number of devices involved in the aggregation as in Section 7.3 above. The results are shown in the top three lines of Figure 7 – notice that even five epochs of cached state offer a significant increase in the number of nodes counted in any aggregate, and that 15 rounds increases the number of nodes involved in the diameter 50 network to 70% (versus less than 10% without a cache). Aside from the temporal smearing described above, there are two additional drawbacks to caching; First, it uses memory that could be used for group storage. Second, it sets a minimum bound on the time that devices must wait before determining their parent has gone offline; given the benefit it provides in terms of accuracy, however, we believe it to be useful despite these disadvantages. The substantial benefit of this technique suggests that allocating RAM to application level caching may be more beneficial than allocating it to lower-level schemes for reliable message delivery, as such schemes cannot take advantage of the semantics of the data being transmitted.

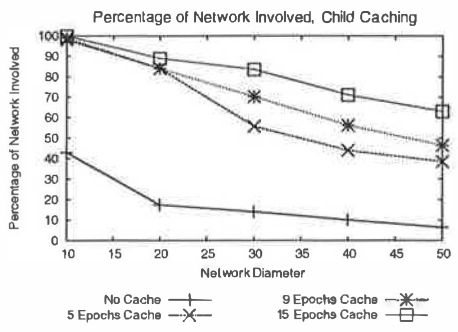


Figure 7: *Percentage of Network Participating in Aggregate For Varying Amounts of Child Cache*

7.5 Using Available Redundancy

Because there may be situations where the RAM or latency costs of the child cache are not desirable, it is worthwhile to look at alternative approaches for improving loss tolerance. In this section, we show how the network topology can be leveraged to increase the quality of aggregates. Consider a mote with two possible choices of parent: instead of sending its aggregate value to just one parent, it can send it to both parents. A node can easily discover that it has multiple parents by building a list of nodes it has heard that are one step closer to the root. Of course, for duplicate-sensitive aggregates (see Section 3.2), sending results to multiple parents has the undesirable effect of causing the node to be counted multiple times. The solution to this is to send part of the aggregate to one parent and the rest to the other. Consider a COUNT; a mote with $c-1$ children and two parents can send a COUNT of $c/2$ to both parents instead of a count of c to a single parent. Generally, if the aggregate can be linearly decomposed in this fashion, it is possible to broadcast just a single message that is received and processed by both parents, so this scheme incurs no message overheads, as long as both parents are at the same level and request data delivery during the same sub-interval of the epoch.

A simple statistical analysis reveals the advantage of doing this: assume that a message is transmitted with probability p, and that losses are independent, so that if a message m from node s is lost in transition to parent P_1, it is no more likely to be lost in transit to P_2. [6] First, consider

[6] Although independent failures are not always a valid assumption,

the case where s sends c to a single parent; the expected value of the transmitted count is $p \times c$ (0 with probability $(1-p)$ and c with probability p), and the variance is $c^2 \times p \times (1-p)$, since these are standard Bernoulli trials with a probability of success p multiplied by a constant c. For the case where s sends $c/2$ to both parents, linearity of expectation tells us the expected value is the sum of the expected value through each parent, or $2 \times p \times c/2 = p \times c$. Similarly, we can sum the variances through each parent:

$$\text{var} = 2 \times (c/2)^2 \times p \times (1-p) = c^2/2 \times p \times (1-p)$$

Thus, the variance of the multiple parent COUNT is much less than with just a single parent, although its expected value is the same. This is because it is much less likely (assuming independence) for the message to both parents to be lost, and a single loss will less dramatically affect the computed value.

We ran an experiment to measure the benefit of this approach in the realistic topology for COUNT with a network diameter of 50. We measured the number of devices involved in the aggregation over a 50 epoch period. When sending to multiple parents, the mean COUNT was 974 ($\sigma = 330$), while when sending to only one parent, the mean COUNT was 94 ($\sigma = 41$). Surprisingly, sending to multiple parents substantially increases the mean aggregate value; most likely this is due to the fact that losses are not truly independent as we assumed above.

This technique applies equally well to any distributive or algebraic aggregate. For holistic aggregates, like MEDIAN, this technique cannot be applied, since partial state records cannot be easily decomposed.

8 Prototype Implementation

Based on the encouraging simulation results presented above, we have built an implementation of TAG for TinyOS Mica motes [19]. The implementation does not currently include many of the optimizations discussed in this paper, but contains the core TAG aggregation algorithm and catalog support for querying arbitrary attributes with simple predicates. In this section, we briefly summarize results from experiments with this prototype, to demonstrate that the simulation numbers given above are consistent with actual behavior and to show that substantial message reductions over a centralized approach are possible in a real implementation.

These experiments involved sixteen motes arranged in a depth four tree, computing a COUNT aggregate over 150 4-second epochs (a 10 minute run.) No child caching or snooping techniques were used. Figure 8 shows the

they will occur when local interference is the cause of loss. For example, a hidden node may garble communication to P_1 but not P_2, or one parent may be in the process of using the radio when the message arrives.

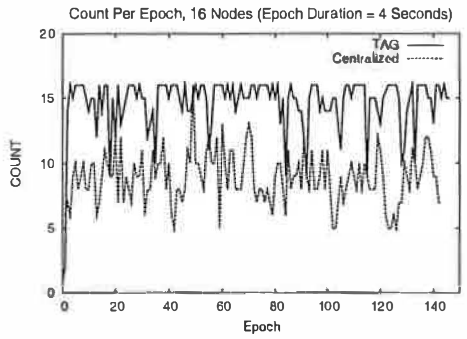


Figure 8: *Comparison of Centralized and TAG based Aggregation Approaches in Lossy, Prototype Environment Computing a COUNT over a 16 node network.*

COUNT observed at the root for a centralized approach, where all messages are forwarded to the root, versus the in network TAG approach. Notice that the quality of the aggregate is substantially better for TAG; this is due to reduced radio contention. To measure the extent of contention and compare the message costs of the two schemes, we instrumented motes to report the number of messages sent and received. The centralized approach required 4685 messages, whereas TAG required just 2330, representing a 50% communications reduction. This is less than the order-of-magnitude shown in Figure 4 for COUNT because our prototype network topology had a higher average fanout than the simulated environment, so messages in the centralized case had to be retransmitted fewer times to reach the root. Per hop loss rates were about 5% in the in network approach. In the centralized approach, increased network contention drove these loss rates to 15%. The poor performance of the centralized case is due to the multiplicative accumulation of loss, such that only 45% of the messages from nodes at the bottom of the routing tree arrived at to the root.

This completes our discussion of algorithms for TAG. We now turn to the extensive related work in the networking and database communities.

9 Related Work

The database community has proposed a number of distributed and push-down based approaches for aggregates in database systems [26, 33], but these universally assume a well-connected, low-loss topology that is unavailable in sensor networks. None of these systems present techniques for loss tolerance or power sensitivity. Furthermore, their notion of aggregates is not tied to a taxonomy, and so techniques for transparently applying various aggregation and routing optimizations are lacking. The partial preaggregation techniques [15] used to enable group eviction were proposed as a technique to deal with very large numbers of groups to improve the efficiency of hash joins and other bucket-based database operators.

The first three components of the partial-state dimension of the taxonomy presented in Section 3.2 (e.g. algebraic, distributive, and holistic) were originally developed as a part of the research on data-cubes [9]; the duplicate sensitivity, exemplary vs. summary, and monotonicity dimensions, as well as the unique and content-sensitive state components of partial-state are our own addition. [29] discusses online aggregation [11] in the context of nested-queries; it proposes optimizations to reduce tuple-flow between outer and inner queries that bear similarities to our technique of pushing `HAVING` clauses into the network.

With respect to query language, our epoch based approach is related to languages and models from the Temporal Database literature; see [27] for a survey of relevant work. The Cougar project at Cornell [23] discusses queries over sensor networks, as does our own work on Fjords [18], although the former only considers moving selections (not aggregates) into the network and neither presents specific algorithms for use in sensor networks.

Literature on active networks [30] identified the idea that the network could simultaneously route and transform data, rather than simply serving as an end-to-end data conduit. The recent SIGCOMM paper on ESP [4] provides a constrained framework for in network aggregation-like operations in a traditional network. Within the sensor network community, work on networks that perform data analysis is largely due to the USC/ISI and UCLA communities. Their work on directed diffusion [13] discusses techniques for moving specific pieces of information from one place in a network to another, and proposes aggregation-like operations that nodes may perform as data flows through them. Their work on low-level-naming[10] proposes a scheme for imposing names onto related groups of devices in a network, in much the way that our scheme partitions sensor networks into groups. Work on greedy aggregation [12] discusses networking protocols for routing data to improve the extent to which data can be combined as it flows up a sensor network – it provides low level techniques for building routing trees that could be useful in computing TAG style aggregates.

These papers recognize that aggregation dramatically reduces the amount of data routed through the network but present application-specific solutions that, unlike the declarative query approach approach of TAG, do not offer a particularly simple interface, flexible naming system, or any generic aggregation operators. Because aggregation is viewed as an application-specific operation in diffusion, it must always be coded in a low-level language. Although some TAG aggregates may also be application-specific, we ask that users provide certain functional guarantees, such as composability with other aggregates, and a classification of semantics (quantity of partial state, monotonicity, etc.) which enable transparent application of various optimizations and create the possibility of a library of common aggregates that TAG users can freely apply within their queries. Furthermore, directed diffusion puts aggregation APIs in the routing layer, so that expressing aggregates requires thinking about how data will be collected, rather than just what data will be collected. This is similar to old-fashioned query processing code that thought about navigating among records in the database – by contrast, our goal is to separate the expression of aggregation logic from the details of routing. This allows users to focus on application issues and enables the system to dynamically adjust routing decisions using general (taxonomic) information about each aggregation function.

Networking protocols for routing data in wireless networks are very popular within the literature [14, 1, 8], however, none of them address higher level issues of data processing, merely techniques for data routing. Our tree-based routing approach is clearly inferior to these approaches for peer to peer routing, but works well for the aggregation scenarios we are focusing on. Work on (N)ACKs (and suppression thereof) in scalable, reliable multicast trees [6, 17] bears some similarity to the problem of propagating an aggregate up a routing tree in TAG. These systems, however, consider only fixed, limited types of aggregates (e.g. ACKs or NAKs for regions or recovery groups.) Finally, we presented an early version of this work in a workshop publication [20].

10 Conclusions

In summary, we have shown how declarative aggregate queries can be distributed and efficiently executed over sensor networks. Our in network approach can provide an order of magnitude reduction in bandwidth consumption over approaches where data is aggregated and processed centrally. The declarative query interface allows end-users to take advantage of this benefit for a wide range of aggregate operations without having to modify low-level code or confront the difficulties of topology construction, data routing, loss tolerance, or distributed computing. Furthermore, this interface is tightly integrated with the network, enabling transparent optimizations that further decrease message costs and improve tolerance to failure and loss.

We plan to extend this work as the data collection needs of the wireless sensor community evolve. We are moving towards an event-driven model where queries can be initiated and results collected in response to external events in the interior of the network, with the results of those internal sub-queries being aggregated across nodes and shipped to points on the network edge.

As sensor networks become more widely deployed, especially in remote, difficult to administer locations, bandwidth- and power-sensitive methods to extract data from those networks will become increasingly important. In such scenarios, the users are often scientists who lack fluency in embedded software development but are interested in using sensor networks to further their own research. For such users, high-level programming interfaces are a necessity; these interfaces must balance simplicity, expressiveness, and efficiency in order to meet data collection and battery lifetime requirements. Given this balance, we see TAG as a very promising service for data collection: the simplicity of declarative queries, combined with the ability of TAG to efficiently optimize and execute them makes it a good choice for a wide range of sensor network data processing situations.

Acknowledgments

Thanks to our shepherd, Deborah Estrin, and to our anonymous reviewers for their thoughtful reviews and advice. Robert Szewczyk, David Culler, Alec Woo, and Ramesh Govindan contributed to the design of the networking protocols discussed in this paper. Per Åke Larson suggested the use of partial preaggregation for group eviction. Kyle Stanek implemented isobar aggregates in our simulator.

References

[1] W. Adjue-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *ACM SOSP*, December 1999.

[2] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *VLDB*, 1987.

[3] D. Barbará, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey data reduction report. *Data Engineering Bulletin*, 20(4):3–45, 1997.

[4] K. Calvert, J. Griffioen, and S. Wen. Lightweight network support for scalable end-to-end services. In *ACM SIGCOMM*, 2002.

[5] A. Cerpa, J. Elson, D.Estrin, L. Girod, M. Hamilton, , and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In *ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, 2001.

[6] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicase framework for light-weight sessions and application level framing. *IEEE Transactions on Networking*, 5(6):784–803, 1997.

[7] D. Ganesan. Network dynamics in rene motes. PowerPoint Presentation, January 2002.

[8] T. Goff, N. Abu-Ghazaleh, D. Phatak, and R. Kahvecioglu. Preemptive routing in ad hoc networks. In *ACM MobiCom*, July 2001.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, February 1996.

[10] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *SOSP*, October 2001.

[11] J. Hellerstein, P. Hass, and H. Wang. Online aggregation. In *Proceedings of the ACM SIGMOD*, pages 171–182, Tucson, AZ, May 1997.

[12] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of network density on data aggregation in wireless sensor networks. Submitted for Publication, ICDCS-22, November 2001.

[13] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MobiCOM*, Boston, MA, August 2000.

[14] J. Kulik, W. Rabiner, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *MobiCOM*, 1999.

[15] P.-Å. Larson. Data reduction by partial preaggregation. In *ICDE*, 2002.

[16] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. Submitted for Publication.

[17] J. Lin and S. Paul. RMTP: A Reliable Multicast Transport Protocol. In *INFOCOM*, pages 1414–1424, 1996.

[18] S. Madden and M. J. Franklin. Fjording the stream: An architechture for queries over streaming sensor data. In *ICDE*, 2002.

[19] S. Madden, W. Hong, J. Hellerstein, and M. Franklin. TinyDB web page. http://telegraph.cs.berkeley.edu/tinydb.

[20] S. Madden, R. Szewczyk, M. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.

[21] A. Mainwaring, J. Polastre, R. Szewczyk, and D. Culler. Wireless sensor networks for habitat monitoring. In *ACM Workshop on Sensor Networks and Applications*, 2002.

[22] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM*, 1997.

[23] P.Bonnet, J.Gehrke, and P.Seshadri. Towards sensor database systems. In *Conference on Mobile Data Management*, January 2001.

[24] C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. In *Workshop on Mobile Computing and Systems Applications*, 1999.

[25] G. Pottie and W. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51 – 58, May 2000.

[26] A. Shatdal and J. Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD*, 1995.

[27] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publisher, 1995.

[28] L. Subramanian and R. H.Katz. An architecture for building self-configurable systems. In *MobiHOC*, Boston, August 2000.

[29] K.-L. Tan, C. H. Goh, and B. C. Ooi. Online feedback for nested aggregate queries with multi-threading. In *VLDB*, 1999.

[30] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survery of active network research. *IEEE Communications*, 1997.

[31] UC Berkeley. Smart buildings admit their faults. Web Page, November 2001. Lab Notes: Research from the College of Engineering, UC Berkeley. http://coe.berkeley.edu/labnotes/1101.smartbuildings.html.

[32] A. Woo and D. Culler. A transmission control scheme for media access in sensor networks. In *ACM Mobicom*, July 2001.

[33] W. P. Yan and P. Å. Larson. Eager aggregation and lazy aggregation. In *VLDB*, 1995.

[34] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *IEEE Infocom*, 2002.

[35] A. Yu and J. Chen. *The POSTGRES95 User Manual*. UC Berkeley, 1995.

Fine-Grained Network Time Synchronization using Reference Broadcasts

Jeremy Elson, Lewis Girod and Deborah Estrin
Department of Computer Science, University of California, Los Angeles
{jelson,girod,destrin}@cs.ucla.edu

Abstract

Recent advances in miniaturization and low-cost, low-power design have led to active research in large-scale networks of small, wireless, low-power sensors and actuators. Time synchronization is critical in sensor networks for diverse purposes including sensor data fusion, coordinated actuation, and power-efficient duty cycling. Though the clock accuracy and precision requirements are often stricter than in traditional distributed systems, strict energy constraints limit the resources available to meet these goals.

We present *Reference-Broadcast Synchronization*, a scheme in which nodes send reference beacons to their neighbors using physical-layer broadcasts. A reference broadcast does not contain an explicit timestamp; instead, receivers use its arrival time as a point of reference for comparing their clocks. In this paper, we use measurements from two wireless implementations to show that removing the sender's nondeterminism from the critical path in this way produces high-precision clock agreement ($1.85 \pm 1.28\mu$sec, using off-the-shelf 802.11 wireless Ethernet), while using minimal energy. We also describe a novel algorithm that uses this same broadcast property to federate clocks across broadcast domains with a slow decay in precision ($3.68 \pm 2.57\mu$sec after 4 hops). RBS can be used without external references, forming a precise relative timescale, or can maintain microsecond-level synchronization to an external timescale such as UTC. We show a significant improvement over the Network Time Protocol (NTP) under similar conditions.

1 Introduction

Time synchronization is a critical piece of infrastructure for any distributed system. For years, NTP (the Network Time Protocol) [24] has kept the Internet's clocks ticking in phase. However, a new class of networks is emerging. Recent advances in miniaturization and low-cost, low-power design have led to active research in large-scale networks of small, wireless, low-power sensors and actuators [1]. These networks require that time be synchronized more precisely than in traditional Internet applications—sometimes on the order of 1μsec—due to their close coupling with the physical world and their energy constraints. For example, precise time is needed to measure the time-of-flight of sound [9, 22]; distribute a beamforming array [34]; form a low-power TDMA radio schedule [2]; integrate a time-series of proximity detections into a velocity estimate [4]; or suppress redundant messages by recognizing duplicate detections of the same event by different sensors [13]. In addition to these domain-specific requirements, sensor network applications often rely on synchronization as typical distributed systems do: for secure cryptographic schemes, coordination of future action, ordering logged events during system debugging, and so forth.

Many network synchronization algorithms have been proposed over the years, but most share the same basic design: a server periodically sends a message containing its current clock value to a client. Our work explores a form of time synchronization that differs from the traditional model. The fundamental property of our design is that it *synchronizes a set of receivers with one another*, as opposed to traditional protocols in which senders synchronize with receivers. In our scheme, nodes periodically send beacon messages to their neighbors using the network's physical-layer broadcast. Recipients use the message's arrival time as a point of reference for comparing their clocks. The message contains no explicit timestamp, nor is it important exactly when it is sent. We call this scheme *Reference-Broadcast Synchronization*, or RBS.

In this paper, we use measurements from two wireless implementations to show that removing the sender's nondeterminism from the critical path in this way results in a dramatic improvement in synchronization over using NTP. We also present an algorithm that allows time to be propagated across broadcast domains without losing the reference-broadcast property. In this way, nodes in a multi-hop network can form a highly precise relative timescale, or maintain microsecond-level synchronization to an external timescale such as UTC.

The most significant limitation of RBS is that it requires a network with a physical broadcast channel. It can not be used, for example, in networks that employ point-to-point links. However, it is applicable for a wide range of applications in both wired and wireless networks where a broadcast domain exists and higher-precision or lower-energy synchronization is required than what NTP can typically provide.

The organization of this paper is as follows: We first review related work in Section 2. In Section 3, we describe the design of traditional time synchronization protocols in more detail, and contrast it to the RBS design philosophy. In Section 4, we describe the basic building blocks of RBS, including estimation of phase offset (§4.1) and clock skew (§4.2). We also present empirical data from two wireless implementations of single-hop RBS (§4.3–§4.5). In Section 5, we present a novel algorithm for extending RBS across broadcast domains with slow precision decay. We also describe empirical results from a multi-hop RBS implementation (§5.3). In Section 6, we describe applications from three research groups that use RBS. Finally, we offer our conclusions and describe our plans for future work in Section 7.

2 Related Work

A landmark paper in computer clock synchronization is Lamport's work that elucidates the importance of *virtual clocks* in systems where causality is more important than absolute time [18]. Though Lamport's work focused on giving events a total order rather than quantifying the time difference between them, it has emerged as an important influence in sensor networks. Many sensor applications require only relative time, for example timing the propagation delay of sound [9], and thus absolute time may not be needed.

Over the years, many protocols have been designed for maintaining synchronization of physical clocks over computer networks [5, 10, 24, 30]. These protocols all have basic features in common: a simple connectionless messaging protocol; exchange of clock information among clients and one or more servers; methods for mitigating the effects of nondeterminism in message delivery and processing; and an algorithm on the client for updating local clocks based on information received from a server. They do differ in certain details: whether the network is kept internally consistent or synchronized to an external standard; whether the server is considered to be the canonical clock, or merely an arbiter of client clocks, and so on.

Mills' NTP [24] stands out by virtue of its scalability, self-configuration in large multi-hop networks, robustness to failures and sabotage, and ubiquitous deployment. NTP allows construction of a hierarchy of time servers, multiply rooted at canonical sources of external time.

Synchronization to an external timescale is typically provided by the Global Positioning System (GPS), a constellation of satellites operated by the U.S. Department of Defense [16]. Commercial GPS receivers can achieve accuracy of better than 200nsec relative to UTC [20]. However, GPS requires a clear sky view, which is unavailable in many application areas—for example, inside of buildings, beneath dense foliage, underwater, on Mars, or behind enemy lines where GPS is jammed. In addition, receivers can require several minutes of settling time, and may be too large, costly, or high-power to justify on a small, cheap sensor node.

Perhaps most closely related to our work are the CesiumSpray system [32] (the foundations of which were described by Veríssimo and Rodrigues [31]), and the 802.11-based broadcast synchronization described by Mock *et al.* [27]. Their systems, like ours, make use of the inherent properties of broadcast media to achieve superior precision. However, their methods require all nodes to lie within a single broadcast domain; multiple domains can not be federated, except by depending on an out of band common view of time (e.g., GPS). In contrast, RBS incorporates a novel multi-hop algorithm; in Section 5, we show it can maintain tight time synchronization across many hops through a wireless network without external infrastructure support. In addition, we have fully decoupled the sender from the receiver—only synchronizing receivers with one another, even when relating the timescale to an external timescale such as UTC. CesiumSpray retains a dependence on the relationship between sender and receiver when synchronizing nodes to UTC.

Liao *et al.* describe a system for synchronizing interfaces on a system-area network, Myrinet, with microsecond accuracy [19]. Although the precision bound in this system is similar to ours, their results depend on the use of an underlying network that offers latency and determinism guarantees in fixed topologies. In contrast, our scheme works over a much broader class of networks, over a much wider area, and does not require a tight coupling between the sender and its network interface.

Similarly, on Berkeley Mote hardware, Hill *et al.* report 2μsec precision for receivers within a single broadcast domain [11]; Ganeriwal *et al.* report 25μsec-per-hop precision across multiple hops [7]. Both of these results are achieved by tightly integrating the MAC with the application, and building a deterministic bit-detector into the MAC layer. RBS does not require that the application be collapsed into the MAC, and indeed will be shown in Section 4.4 to work on commodity hardware (802.11). However, their work is complementary to ours, as our focus is not on building deterministic detectors, but rather making the best use of existing detectors. By leveraging Hill's 0.25μsec-precision bit detector, RBS could achieve several times the precision of their scheme, based on their 2μsec jitter budget analysis.

3 Traditional Synchronization Methods

Before discussing RBS in detail, we describe how existing time synchronization protocols typically work, and their usual sources of error.

Many synchronization protocols exist, and most share a basic design: a server periodically sends a message containing its current clock value to a client. A simple one-way message suffices if the typical latency from server to client is small compared to the desired accuracy. A common extension is to use a client request followed by a server's response. By measuring the total round-trip-time of the two packets, the client can estimate the one-way latency. This allows for more accurate synchronization by accounting for the time that elapses between the server's creation of a timestamp and the client's reception of it. NTP is a ubiquitously adopted protocol for Internet time synchronization that exemplifies this design.

3.1 Sources of Time Synchronization Error

The enemy of precise network time synchronization is *non-determinism.* Latency estimates are confounded by random events that lead to asymmetric round-trip message delivery delays; this contributes directly to synchronization error. To better understand the source of these errors, it is useful to decompose the source of a message's latency. Kopetz and Schwabl characterize it as having four distinct components [17]:

- *Send Time*—the time spent at the sender to construct the message. This includes kernel protocol processing and variable delays introduced by the operating system, e.g. context switches and system call overhead incurred by the synchronization application. Send time also accounts for the time required to transfer the message from the host to its network interface.

- *Access Time*—delay incurred waiting for access to the transmit channel. This is specific to the MAC protocol in use. Contention-based MACs (e.g., Ethernet [23]) must wait for the channel to be clear before transmitting, and retransmit in the case of a collision. Wireless RTS/CTS schemes such as those in 802.11 networks [14] require an exchange of control packets before data can be transmitted. TDMA channels [29] require the sender to wait for its slot before transmitting.

- *Propagation Time*—the time needed for the message to transit from sender to receivers once it has left the sender. When the sender and receiver share access to the same physical media (e.g., neighbors in an ad-hoc wireless network, or on a LAN), this time is very small as it is simply the physical propagation time of the message through the media. In contrast, Propagation Time dominates the delay in wide-area networks, where it includes the queuing and switching delay at each router as the message transits through the network.

- *Receive Time*—processing required for the receiver's network interface to receive the message from the channel and notify the host of its arrival. This is typically the time required for the network interface to generate a message reception signal. If the arrival time is timestamped at a low enough level in the host's operating system kernel (e.g., inside of the network driver's interrupt handler), the Receive Time does not include the overhead of system calls, context switches, or even the transfer of the message from the network interface to the host.

Existing time synchronization algorithms vary primarily in their methods for estimating and correcting for these sources of error. For example, Cristian observed that performing larger numbers of request/response experiments will make it more likely that at least one trial will not encounter random delays [5]. This trial, if it occurs, is easily identifiable as the shortest round-trip time. NTP filters its data using a variant of this heuristic.

Our scheme takes a different approach to reducing error. Instead of estimating error, we exploit the broadcast channel available in many physical-layer networks to remove as much of it as possible from the critical path. Our contributions in this paper stem from the observation that a message that is broadcast at the physical layer will arrive at a set of receivers with very little variability in its delay. Although the Send Time and Access Time may be unknown, and highly variable from message to message, the nature of a broadcast dictates that for a *particular* message, these quantities are *the same for all receivers*. This observation was also made by Veríssimo and Rodrigues [31], and later became central to their CesiumSpray system [32].

The fundamental property of Reference-Broadcast Synchronization is that a broadcast message is only used to *synchronize a set of receivers with one another*, in contrast with traditional protocols that synchronize the sender of a message with its receiver. Doing so removes the Send Time and Access Time from the critical path, as shown in Figure 1. This is a significant advantage for synchronization on a LAN, where the Send Time and Access time are typically the biggest contributors to the nondeterminism in the latency. Mills attributes most of the phase error seen when synchronizing an NTP client workstation to a GPS receiver on the same LAN (500μsec–2000μsec in his 1994 study [25]) to these factors—Ethernet jitter and collisions.

To counteract these effects, an RBS broadcast is always used as a *relative* time reference, never to communicate an absolute time value. It is exactly this property that eliminates error introduced by the Send Time and Access Time: each receiver is synchronizing to a reference packet which was, by definition, injected into the physical channel at the same instant. The message itself does not contain a timestamp generated by the sender, nor is it important exactly when it is sent. In fact, the broadcast does not even need to be a dedicated timesync packet. Almost any extant broadcast can be used to recover timing information—for example, ARP packets in Ethernet, or the broadcasted control traffic in wireless networks (e.g., RTS/CTS exchanges or route discovery packets).

3.2 Characterizing the Receiver Error

Previously, we described the four sources of latency in sending a message. Two of these—the Send Time and Access Time—are dependent on factors such as the instantaneous load on the sender's CPU and the network. This makes them the most nondeterministic and difficult to estimate. As described above, RBS eliminates the effect of these error sources altogether; the two remaining factors are the Propagation Time and Receive Time.

For our purposes, we consider the Propagation Time to be effectively 0. The propagation speed of electromagnetic signals through air is close to c,[1] and through copper wire about $\frac{2}{3}c$. For a LAN or ad-hoc network spanning tens of feet, propagation time is at most tens of nanoseconds, which does not contribute significantly to our μsec-scale error budget. (In sensor networks, the error budget is often driven by the need to sense phenomena, such as sound, that move much more slowly than the RF-pulse used to synchronize the observers.) In addition, RBS is only sensitive to the *difference* in propagation time between a pair of receivers, as depicted in Figure 1.

The length of a bit gives us an idea of the Receive Time. For a receiver to interpret a message at all, it must be synchronized to the incoming message within one bit time. Latency due to processing in the receiver electronics is irrelevant as long as it is *deterministic*, since our scheme is only sensitive to *differences* in the receive time of messages within a set of receivers.[2] In addition, the system clock can easily be read at interrupt time when a packet is received; this removes delays due to receiver protocol processing, context switches, and interface-to-host transfer from the critical path.

To verify our assumptions about expected receiver error, we turned to our laboratory's wireless sensor network testbed [4]. Specifically, we tested the Berkeley Mote, a postage-stamp sized narrowband radio and sensor platform developed by Pister *et al.* at Berkeley [15]. Motes use a minimal event-based operating system developed by Hill *et al.* specifically for that hardware platform called TinyOS [12]. We programmed 5 Motes to raise a GPIO pin high upon each packet arrival, and attached those signal outputs to an external logic analyzer that recorded the time of the packet reception events. An additional Mote emitted 160 broadcast packets over

[1] A convenient rule of thumb is 1*nsec*/foot.

[2] It is important to consider effects of any intentional nondeterminism on the part of a receiver, such as aggregation of packets in a queue before raising an interrupt.

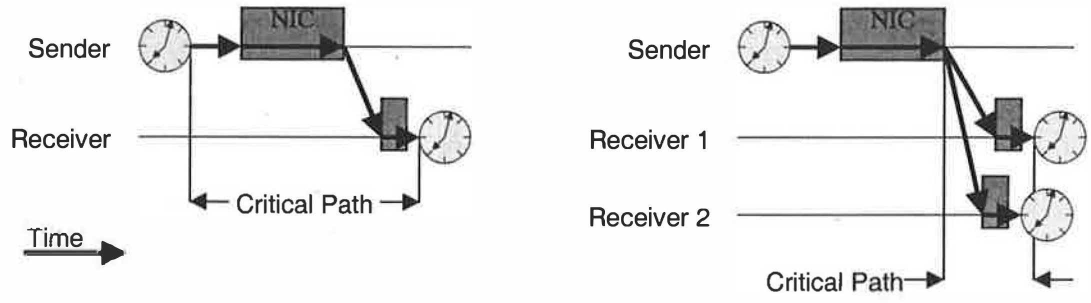


Figure 1: A critical path analysis for traditional time synchronization protocols (*left*) and RBS (*right*). For traditional protocols working on a LAN, the largest contributions to nondeterministic latency are the Send Time (from the sender's clock read to delivery of the packet to its NIC, including protocol processing) and Access Time (the delay in the NIC until the channel becomes free). The Receive Time tends to be much smaller than the Send Time because the clock can be read at interrupt time, before protocol processing. In RBS, the critical path length is shortened to include only the time from the injection of the packet into the channel to the last clock read.

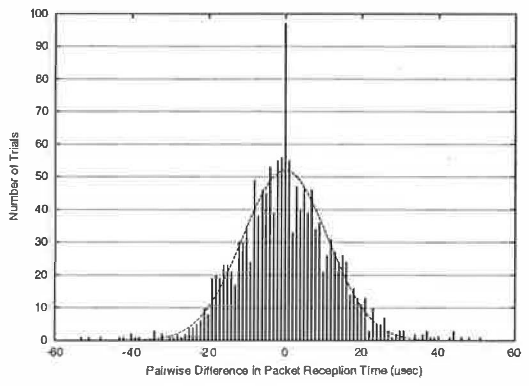


Figure 2: A histogram showing the distribution of inter-receiver phase offsets recorded for 1,478 broadcast packets, grouped into 1μsec buckets. The curve is a plot of the best-fit Gaussian parameters. ($\mu = 0, \sigma = 11.1\mu$sec, confidence=99.8%)

the course of 3 minutes, with random inter-packet delays ranging from 200msec to 2 seconds. For each pulse transmitted, we computed the difference in the pulse's packet reception time for each of the 10 possible pairings of the 5 receivers. Some pulses were lost at some receivers due to the normal vagaries of RF links. A total of 1,478 pairings were computed.

A histogram showing the distribution of the receivers' phase offsets is shown in Figure 2. The maximum phase error observed in any trial was 53.4μsec. The Mote's radios operate at 19,200 baud and thus have a bit time of 52μsec. This correspondence supports our reasoning that a receiver's jitter is unlikely to be much greater than a single bit time, as long as the receiver electronics have a deterministic delay between packet reception and host notification.

The phase offset distribution appears Gaussian; the chi-squared test indicates 99.8% confidence in the best fit parameters—$\mu = 0, \sigma = 11.1\mu$sec. This is useful for several reasons. As we will see in later sections, a well-behaved distribution enables us to significantly improve on the error bound statistically, by sending multiple broadcast packets. In addition, Gaussian receivers are easy to model realistically in numeric analyses. This facilitates quick experimentation with many different algorithms or scenarios whose systematic exploration is impractical, such as examining the effects of scale.

4 Reference-Broadcast Synchronization

So far, we characterized the determinism of our receiver hardware. We now turn to the question of using that receiver to achieve the best possible clock synchronization.

We should note that our research does not encompass the question of how to build a more deterministic receiver. This is best answered by those who design them. Plans for forthcoming revisions of the Mote's TinyOS are reported to include better phase lock between a receiver and incoming message. Instead, we ask: How well can we synchronize clocks with a *given* receiver? How is the precision we can achieve related to that receiver's jitter?

In this section, we build up the basic RBS algorithm for nodes within a single broadcast domain. (Applying RBS in multihop networks will be considered in Section 5.)

First, we consider how RBS can be used to estimate the relative phase offsets among a group of receivers. Next, we describe a slightly more complex scheme that also corrects for clock skew. We also describe implementation of RBS on both Berkeley Motes and commodity hardware, and quantify the performance of RBS vs. NTP. Finally, we describe how the combination may be used to achieve *post-facto* synchronization.

4.1 Estimation of Phase Offset

The simplest form of RBS is the broadcast of a single pulse to two receivers, allowing them to estimate their relative phase offsets. That is:

1. A transmitter broadcasts a reference packet to two receivers (i and j).
2. Each receiver records the time that the reference was received, according to its local clock.
3. The receivers exchange their observations.

Based on this single broadcast alone, the receivers have sufficient information to form a local (or *relative*) timescale. Global (or *absolute*) timescales such as UTC are important, and we will see in Section 5.4 how RBS can be extended to provide them. However, in this section we will first consider construction of local timescales—which are often sufficient for sensor network applications. For example, if node i at position (0, 0) detects a target at time $t = 4$, and j at position (0, 10) detects it at $t = 5$, we can conclude that the target is moving north at 10 units per second. This comparison does not require absolute time synchronization; the reference broadcast makes it possible to compare the time of an event observed by i with one observed by j.[3]

The performance of this scheme is closely related to three factors:

1. The number of receivers that we wish to synchronize (n). We assumed above that $n = 2$; however, collaborative applications often require synchronization of many nodes simultaneously. As the set grows, the likelihood increases that at least one will be poorly synchronized. We define the *group dispersion* as the maximum phase error between any of the $\binom{n}{2}$ pairs of receivers in a group.
2. The nondeterminism of the underlying receiver, as we discussed in Section 3.2.
3. Clock skew in the receivers, as their oscillators all tick at different rates. A single reference will provide only *instantaneous* synchronization. Immediately afterward, the synchronization will degrade as the clocks drift apart. Precision will therefore decay as more time elapses between the synchronization pulse and the event of interest. Typical crystal oscillators are accurate on the order of one part in 10^4 to 10^6 [33]—that is, two nodes' clocks will drift 1–100μsec per second.

[3]Our prototype implementation never sets the local clock based on reference broadcasts; instead, it provides a library function for timestamp conversion. That is, it can convert a time value that was generated by the local clock to the value that would have been generated on another node's clock at the same time, and vice-versa.

We will see in the next section how clock skew can be estimated. However, assume for the moment that we already know the clock skew and have corrected for it. Let us instead consider what is required to correct for the nondeterminism in the receiver. Recall from Figure 2 that we empirically observed the Mote's receiver error to be Gaussian. We can therefore increase the precision of synchronization statistically, by sending more than one reference:

1. A transmitter broadcasts m reference packets.
2. Each of the n receivers records the time that the reference was observed, according to its local clock.
3. The receivers exchange their observations.
4. Each receiver i can compute its phase offset to any other receiver j as the average of the phase offsets implied by each pulse received by both nodes i and j. That is, given

   n: the number of receivers

   m: the number of reference broadcasts, and

   $T_{r,b}$: r's clock when it received broadcast b,

$$\forall i \in n, j \in n: \text{Offset}[i,j] = \frac{1}{m}\sum_{k=1}^{m}(T_{j,k} - T_{i,k}). \quad (1)$$

Because this basic scheme does not yet account for clock skew (a feature added in §4.2), it is not yet implementable on real hardware. We therefore turned to a numeric simulation based on the receiver model that we derived empirically in Section 3.2. In each trial, n nodes were given random clock offsets. m pulse transmission times were selected at random. Each pulse was "delivered" to every receiver by timestamping it using the receiver's clock, including a random error matching the Gaussian distribution parameters shown in Figure 2 ($\sigma = 11.1$). An offset matrix was computed as given in Equation 1. Each of these $O(n^2)$ computed offsets was then compared with the "real" offsets; the maximum difference was considered to be the *group dispersion*. We performed this analysis for values of $m = 1 \ldots 50$ and $n = 2 \ldots 20$. At each value of m and n, we performed 1,000 trials and calculated the mean group dispersion, and standard deviation from the mean. The results are shown in Figure 3.

This numeric simulation suggests that in the simplest case of 2 receivers (the lower curve of Figure 3a), 30 reference broadcasts can improve the precision from 11μsec to 1.6μsec, after which we reach a point of diminishing return.[4] In a group of 20 receivers, the dispersion can be reduced to 5.6μsec. Figure 3b shows a 3D view of the same dataset for all receiver group sizes from 2 to 20. This view also shows that larger numbers of broadcasts significantly mitigate the effect of larger group size on precision decay.

4.2 Estimation of Clock Skew

So far, we have described a method for estimating phase offset assuming that there was no clock skew—that is, that all the nodes' clocks are running at the same rate. Of course, this is not a realistic assumption. A complete description of crystal oscillator behavior is beyond the scope of this paper;[5] however, to a first approximation, the important characteristics of an oscillator are:

- Accuracy—the agreement between an oscillator's expected and actual frequencies. The difference is the *frequency error*; its maximum is usually specified by the manufacturer. The crystal oscillators found in most consumer electronics are accurate to one part in 10^4 to 10^6.
- Stability—An oscillator's tendency to stay at the same frequency over time. Frequency stability can be further subdivided into *short-term* and *long-term* frequency stability. Short-term instability is primarily due to environmental factors, such as variations in temperature, supply voltage, and shock. Long-term instability results from more subtle effects, such as oscillator aging.

[4] 11μsec is the expected average result for a single pulse delivered to two receivers; this is the standard deviation of the inter-receiver phase error, found in Section 3.2, upon which the analysis was based.

[5] [33] has a good introduction to the topic and pointers to a more comprehensive bibliography.

The phase difference between two nodes' clocks will change over time due to frequency differences in the oscillators. The basic algorithm that we described earlier is not physically realizable without an extension that takes this into account—in the time needed to observe 30 pulses, the phase error between the clocks will have changed.

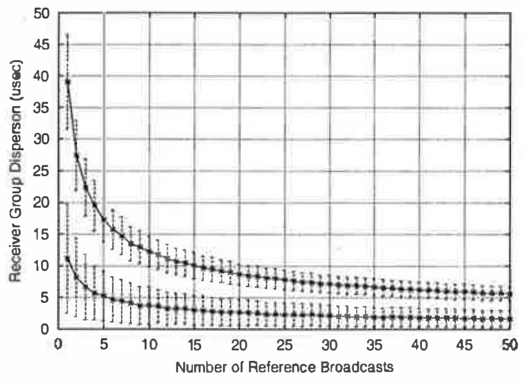
Complex disciplines exist that can lock an oscillator's phase and frequency to an external standard [26]. However, we selected a very simple yet effective algorithm to correct skew. Instead of *averaging* the phase offsets from multiple observations, as shown in Equation 1, we perform a *least-squares linear regression*. This offers a fast, closed-form method for finding the best fit line through the phase error observations over time. The frequency and phase of the local node's clock with respect to the remote node can be recovered from the slope and intercept of the line.

Of course, fitting a *line* to these data implicitly assumes that the frequency is stable, i.e., that the phase error is changing at a constant rate. As we mentioned earlier, the frequency of real oscillators will change over time due, for example, to environmental effects. In general, network time synchronization algorithms (e.g., NTP) correct a clock's phase and its oscillator's frequency error, but do not try to model its frequency instability. That is, frequency adjustments are made continuously based on a recent window of observations relating the local oscillator to a reference. Our prototype RBS system is similar; it models oscillators as having high short-term frequency stability by ignoring data that is more than a few minutes old.

4.3 Implementation on Berkeley Motes

To test these algorithms, we first built a prototype RBS system based around the Berkeley Motes we described in Section 3.2. In the first configuration we tested, 5 Motes periodically broadcasted a reference pulse with a sequence number. Each of them used a 2μsec resolution clock to timestamp the reception times of incoming broadcasts. We then performed an off-line analysis of the data. Figure 4a shows the results of a typical experiment after automatic rejection of outliers (described below). Each point is the difference between the times that the two nodes reported receiving a reference broadcast, plotted on a timescale defined by one node's clock. That is, given

$T_{r,b}$: r's clock when it received broadcast b,

a) 

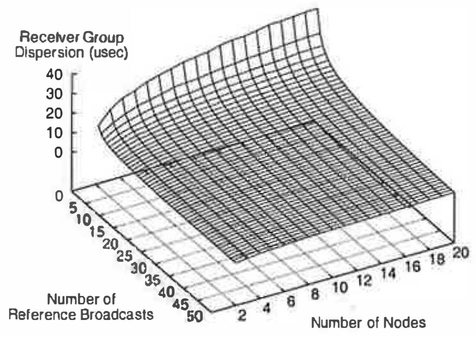
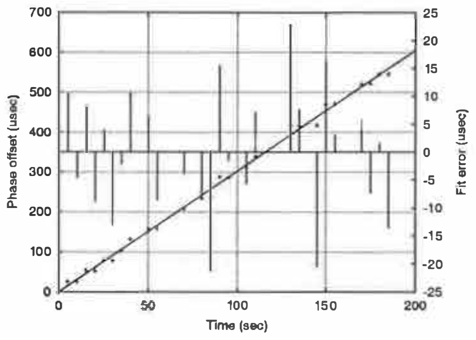
b) 

Figure 3: Analysis of expected group dispersion (i.e., maximum pairwise error) after reference-broadcast synchronization. Each point represents the average of 1,000 simulated trials. The underlying receiver is assumed to report packet arrivals with error conforming to the distribution in Figure 2. *a)* Mean group dispersion, and standard deviation from the mean, for a 2-receiver (*bottom*) and 20-receiver (*top*) group. *b)* A 3D view of the same dataset, from 2 to 20 receivers (inclusive).

a) 

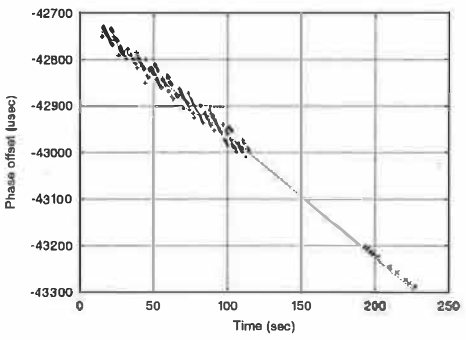
b) 

Figure 4: An analysis of clock skew's effect on RBS. Each point represents the phase offset between two nodes as implied by the value of their clocks after receiving a reference broadcast. A node can compute a least-squared-error fit to these observations (*diagonal lines*), and thus convert time values between its own clock and that of its peer. *a*) Synchronization of the Mote's internal clock. The vertical impulses (read with respect to the *y*2 axis) show the distance of each point from the best-fit line. *b*) Synchronization of clocks on PC104-compatible single-board-computers using a Mote as NIC. The points near $x = 200$ are reference pulses, which show a synchronization error of 7.4μsec 60 seconds after the last sync pulse.

for each pulse k that was received by receivers r_1 and r_2, we plot

$$x = T_{r_1,k}$$
$$y = T_{r_2,k} - T_{r_1,k}$$

For visualization purposes, the data are normalized so that the first pulse occurs at $(0,0)$. The diagonal line drawn through the points represents the best linear fit to the data—i.e., the line that minimizes the RMS error. The vertical impulses, read with respect to the right-hand y axis, show the distance from each point to the best-fit line.

The residual error—that is, the RMS distance of each point to the fit line—gives us a good idea of the quality of the fit. We can reject outliers by discarding the point that contributes most to the total error, if its error is greater than some threshold (e.g. 3σ). In this experiment, the RMS error was 11.2μsec. (A more systematic study of RBS synchronization error is described in later sections.)

The slope of the best-fit line defines the clock skew, when measured by receiver r_1's clock. We can see, for example, that these two oscillators drifted about 300μsec after 100 "r_1 seconds." The line's intercept defines the initial phase offset. When combined, we have sufficient information to convert any time value generated by r_1's clock to the value that would have been generated by

r_2's clock at the same time. We will see in Section 5.4 that "r_2" may also be an external standard such as UTC. However, even the internally consistent mapping is useful in a sensor network, as we saw in Section 4.1.

In a second test configuration, we used Motes as "network interfaces" rather than as stand-alone nodes. Motes were connected via a 9600-baud serial link to PC104-based single-board-computers running the Linux operating system. In some sense, this makes the problem more difficult because we are extending the path between the receiver (the Mote) and the clock to be synchronized (under Linux). We modified the Linux kernel to timestamp serial-port interrupts, allowing us to accurately associate complete over-the-air messages with kernel timestamps.

In this second test, shown in Figure 4b, one "Mote-NIC" sent reference broadcasts to two receivers for two minutes. Each reception was timestamped by the Linux kernel. After one minute of silence, we injected 10 reference pulses into the PC's parallel port; these pulses appear at the right of the figure. We computed the best-fit line based on the Mote-NIC pulses, after automatic outlier rejection. There was a 7.4μsec residual error between the estimate and the reference pulses.

4.4 Commodity Hardware Implementation

The performance of RBS on Berkeley Motes was very encouraging. However, it is reasonable to wonder if its success was due to the algorithm itself or simply the fact that it was implemented on a tightly integrated radio and processor platform. In addition, we were curious about the relative performance of RBS and NTP. To answer these questions—and provide RBS on a platform more generally available to users—we implemented RBS as a UNIX daemon, using UDP datagrams instead of Motes. We used a commodity hardware platform that is part of our testbed: StrongARM-based Compaq IPAQs with Lucent Technologies 11Mbit 802.11 wireless Ethernet adapters. Our IPAQs run the "Familiar" Linux distribution[6] with Linux kernel v2.4. In this test, all the wireless Ethernet adapters are connected to a wireless 802.11 base station.

To make the comparison as fair as possible, we first implemented RBS under the same constraints as NTP: a pure userspace application with no special kernel or hardware support, or special knowledge of the MAC layer (other than that it supports broadcasts). Like NTP, our Linux RBS daemon uses UDP datagrams sent and received via the Berkeley socket interface. Packet reception times are recorded in userspace by using the standard `gettimeofday()` library function (and underlying system call). The daemon records the time after learning that a new packet has arrived via an unblocked `select()` call. The IPAQ's clock resolution is 1μsec.

[6]http://www.familiar.org

Our RBS daemon simultaneously acts in both "sender" and "receiver" roles. Every 10 seconds (slightly randomized to avoid unintended synchronization), each daemon emits a pulse packet with a sequence number and sender ID. The daemon also watches for such packets to arrive; it timestamps them and periodically sends a report of these timestamps back to the pulse sender along with its receiver ID. The pulse sender collects all of the pulse reception reports and computes clock conversion parameters between each pair of nodes that heard its broadcasts. These parameters are then broadcast back to local neighbors. The RBS daemons that receive these parameters make them available to users. RBS never sets the nodes' clocks, but rather provides a user library that converts UNIX `timevals` from one node ID to another.

The clock conversion parameters are the slope and intercept of the least-square linear regression, similar to the examples shown in Figure 4. Each regression is based on a window of the 30 most recent pulse reception reports. In practice, a small number of pulses are outliers, not within the Gaussian mode shown in Figure 2, due to random events such as concurrent serial and Ethernet interrupts. These outliers are automatically rejected based on an adaptive threshold equal to 3 times the median fit error of the set of points not yet rejected. (Early versions used a more traditional "3σ" approach, but the standard deviation was found to be too sensitive to gross outliers.) The remaining points are iteratively re-fit, and the outlier threshold recomputed, until no further points are rejected. If more than half the points are rejected as outliers, the fit fails.

To test the synchronization accuracy, we connected a GPIO output from each of two IPAQs to an external logic analyzer. The analyzer was programmed to report the time difference between two pulses seen on each of its input channels. We wrote a Linux kernel module that accepts a target pulse-time from a userspace application, then emits a GPIO pulse when the local clock indicates that the target time has arrived. By implementing the pulsing service inside the kernel, with interrupts disabled shortly before the target time arrives, there is virtually no phase error between the GPIO pulse and the node's clock. In other words, the kernel module ensures that error between the pulses as seen by the logic analyzer is entirely due to the nodes' clock synchronization error, not an artifact of the test. (This kernel module purely facilitates testing; it was not used to help the synchronization of either NTP or RBS.)

Using this experimental framework, we tested three different synchronization schemes:

- RBS—Our reference broadcast synchronization. A third IPAQ was used as a pulse sender to facilitate synchronization between the two test systems. NTP was off during this test.
- NTP—The standard NTP package, version 4.1.1a, ported to our IPAQs. The `ntpdate` program was first used to achieve gross synchronization. Then, the `ntpd` daemon was run, configured to send synchronization packets every 16 seconds (the maximum frequency it allows). The clients were allowed to synchronize to our lab's stratum 1 GPS-steered NTP server for several hours at this high sampling frequency before the test began. The NTP server was on the wired side of our network (i.e., accessed via the wireless base station).
- NTP-Offset—In our test, RBS has a potentially unfair advantage over NTP. Although NTP maintains a continuous estimate of the local clock's phase error with respect to its reference clock, it also limits the rate at which it is willing to correct this error. This is meant to prevent user applications from seeing large frequency excursions or discontinuities in the timescale. Our initial RBS implementation has no such restriction. Since our test only evaluates phase error, and does not give any weight to frequency stability, it might favor RBS. To eliminate this possible advantage, we created *NTP-Offset* synchronization. This was similar to the NTP method; however, during each trial, the test application queried the NTP daemon (using the `ntpq` program) for its current estimate of the local clock's phase error. This value was subtracted from the test pulse time.

For each of these synchronization schemes, we tested two different traffic scenarios:

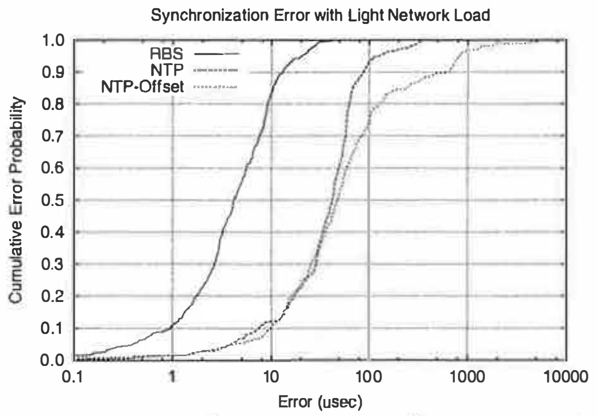
- Light network load—The 802.11 network had very little background traffic other than the minimal load generated by the synchronization scheme itself.
- Heavy network load—Two additional IPAQs were configured as traffic generators. Each IPAQ sent a series of randomly-sized UDP datagrams, each picked uniformly between 500 and 15,000 bytes (IP fragmentation being required for larger packets). The inter-packet delay was 10msec. The cross-traffic was entirely among third parties—that is, the source and destination of this traffic were neither the synchronization servers nor the systems under test. The average aggregate offered load of this cross-traffic was approximately 6.5Mbit/sec.

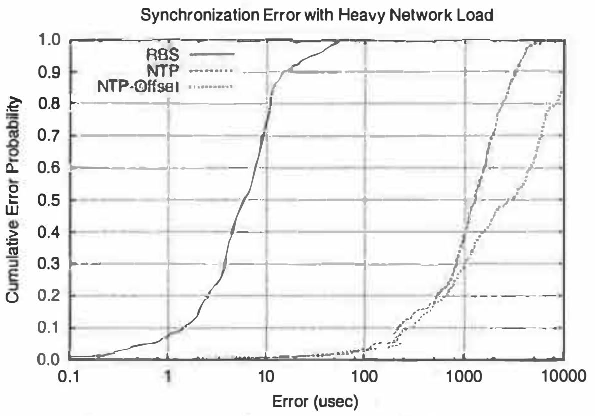
Each of the six test scenarios consisted of 300 trials, with an 8 second delay between each trial, for a total test time of 40 minutes per scenario. The results are shown in Figure 5. In the light traffic scenario, RBS performed more than 8 times better than NTP—an average of 6.29 ± 6.45 μsec error, compared to $51.18 \pm 53.30\mu$sec for NTP. 95% of RBS trials had an error of 20.53μsec or better, compared to a 131.20μsec bound for 95% of NTP trials.

Much of the 6.29μsec error seen with our userspace RBS implementation is due to the jitter in software—the code path through the network stack, the time required to schedule the daemon, and the system calls from userspace to determine the current time. We will see in the next section that significantly better precision can be achieved using packet timestamps acquired inside the kernel's network interrupt handler.

In our heavy traffic scenario, the difference was even more dramatic. As we discussed in Section 3, NTP is most adversely affected by path asymmetries that confound its latency estimate. When the network is under heavy load, it becomes far more likely that the medium access delay will be different during the trip to and from the NTP server. RBS, in contrast, has no dependence on the exact time packets are sent; variable MAC delays have no effect. While NTP suffered more than a 30-fold degradation in precision (average 1,542μsec, 95% within 3,889μsec), RBS was almost completely unaffected (average 8.44μsec, 95% within 28.53μsec). The slight degradation in RBS performance was due to packet loss, which reduced the number of broadcast pulses available for comparison with peers. Several instances of RBS phase excursion were also observed when a packet containing a clock parameter update was lost, forcing clients to continue using aging data.

Surprisingly, the NTP-Offset method almost uniformly performed more poorly than plain NTP. The cause was not clear, but we suspect this was due to the inter-packet jitter in the 802.11 MAC. The low-pass filter built into NTP's clock discipline algorithm was better suited to model the high-jitter network than the more responsive online phase estimator.





Network Load	Synchronization	Mean Error	Std Dev	50% Bound	95% Bound	99% Bound
Light	RBS	6.29	6.45	4.22	20.53	29.61
Light	NTP	51.18	53.30	42.52	131.20	313.64
Light	NTP-Offset	204.17	599.44	48.15	827.42	4334.50
Heavy	RBS	8.44	9.37	5.86	28.53	48.61
Heavy	NTP	1542.27	1192.53	1271.38	3888.79	5577.82
Heavy	NTP-Offset	5139.08	6994.58	3163.11	15156.44	38897.36

Figure 5: Synchronization error for RBS and NTP between two Compaq IPAQs using an 11Mbit 802.11 network. Clock resolution was 1μsec. All units are μsec. "NTP-Offset" uses an NTP-disciplined clock with a correction based on NTP's instantaneous estimate of its phase error; unexpectedly, this correction led to poorer synchronization. RBS performed more than 8 times better than NTP on a lightly loaded network. On a heavily loaded network, NTP further degraded by more than a factor of 30, while RBS was virtually unaffected.

4.5 RBS using 802.11 and Kernel Timestamps

In the previous section, we described the performance of RBS when implemented as a userspace application. This provided a fair comparison with NTP. However, for practical use in our testbed, we give RBS the additional advantage of packet timestamps acquired in the network interface's interrupt handler. Timestamping at interrupt-time, before the packet is even transferred from the NIC, significantly reduces jitter and is a standard feature of the Linux kernel. The metadata is accessible by reading packets using the LBNL packet capture library, `libpcap` [21], instead of the usual socket interface.

Using kernel timestamps, the performance of RBS improved considerably, to $1.85 \pm 1.28\mu$sec (see Figure 8), from $6.29 \pm 6.45\mu$sec in the user-space implementation.

We believe this result was primarily limited by the 1μsec clock resolution of the IPAQ, and that finer precision can be achieved with a finer-resolution clock. In our future work, we hope to try RBS on a such a platform: for example, using the Pentium's TSC, a counter which runs at the frequency of the processor core (e.g., 1GHz).

4.6 Application to Post-Facto Synchronization

Sensor network nodes are wireless and unattended; their finite energy becomes a dominating factor in their design. It is often not feasible to keep the processor or radio powered continuously. Such a "deep sleep" confounds traditional protocols like NTP that keep the clock synchronized at all times. We therefore have previously proposed *post-facto synchronization* [6]. In this scheme, nodes normally stay in a low power state, with *un*synchronized clocks, until a event of interest occurs. Only after such an event are the clocks reconciled. This prevents energy from being wasted on achieving unnecessary synchronization.

An interesting facet of the RBS clock skew estimator is that it is is also an effective form of post-facto synchronization. After an event of interest, nodes can power their radios up and exchange sync pulses until the best-fit line that relates nodes' clocks to each other has been computed satisfactorily (e.g., the RMS error has fallen below some threshold). Our experiment shown in Figure 4b suggests this line can be used to precisely extrapolate backwards, estimating what the phase offset was at a time in the past, such as when the event occurred.

4.7 Summary of Advantages of RBS

So far, we have built up the basic RBS algorithm, showing how reference broadcasts can be used to relate a set of receivers' clocks to one another. Over time, we can estimate both relative phase and skew. On Berkeley Motes, RBS can synchronize clocks within 11μsec, with our relatively slow 19,200 baud radios. On commodity hardware—Compaq IPAQs using an 11 Mbit/sec 802.11 network—we achieved synchronization of $6.29 \pm 6.45\mu$sec, 8 times better than NTP under the same conditions. Using kernel timestamps, precision nearly reached the clock resolution—$1.85 \pm 1.28\mu$sec.

The advantages of RBS include:

- The largest sources of nondeterministic latency can be removed from the critical path by using the broadcast channel to synchronize receivers with one another. This results in significantly better precision synchronization than algorithms that measure round-trip delay.

- Multiple broadcasts allow tighter synchronization because residual errors tend to follow well-behaved distributions. In addition, multiple broadcasts allow estimation of clock skew, and thus extrapolation of past phase offsets. This enables *post-facto synchronization*, saving energy in applications that need synchronized time infrequently and unpredictably.

- Outliers and lost packets are handled gracefully; the best fit line in Figure 4 can be drawn even if some points are missing.

- RBS allows nodes to construct *local* timescales. This is useful for sensor networks or other applications that need synchronized time but may not have an absolute time reference available. Absolute time synchronization can also be achieved, as we will describe in Section 5.4.

The primary shortcoming of RBS as we have described it thus far is that it can only be used to synchronize a set of nodes that lie within a single physical-layer broadcast domain. We address this limitation in the next section.

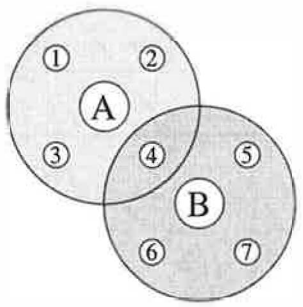


Figure 6: A simple topology where multi-hop time synchronization is required. Nodes A and B send sync pulses, establishing two locally synchronized neighborhoods. Receiver 4 hears both A and B, and can relate nodes in either neighborhood to each other.

5 Multi-Hop Time Synchronization

RBS as we have described it until now has used broadcasts to coordinate a set of receivers that lie within a single broadcast domain. A natural question that arises is how this technique might be extended so as to coordinate receivers whose span is larger than a single physical-layer broadcast. For example, the extent of wireless sensor networks is usually much larger than the broadcast range of any single node. In traditional LANs, the broadcast domain of an Ethernet is limited; LANs are interconnected with routers, bridges, or other gateways that do not propagate broadcasts at the physical layer.

In these scenarios, an extension to basic RBS can be used to synchronize a group of nodes that lie beyond the range of a single broadcast. Consider the example topology shown in Figure 6. The lettered nodes, A and B, both send a sync pulse. A and B can not hear each other, but each of them are heard by 4 receivers. Receivers that are in the same neighborhood (i.e., have heard the same sync pulse) can relate their clocks to each other, as described in previous sections. However, notice that receiver 4 is in a unique position: it can hear the sync pulses from both A *and* B. This allows receiver 4 to relate the clocks in one neighborhood to clocks in the other.

5.1 Multihop Clock Conversion

To illustrate how the conversion works, imagine that we wish to compare the times of two events that occurred at opposite ends of the network—near receivers 1 and 7. Nodes A and B both send synchronization pulses, at times P_A and P_B. Say that receiver 1 observes event

E_1 2 seconds after hearing A's pulse (e.g., $E_1 = P_A + 2$). Meanwhile, receiver 7 observes event E_7 at time $P_B - 4$. These nodes could consult with receiver 4 to learn that A's pulse occurred 10 seconds after B's pulse: $P_A = P_B + 10$. Given this set of constraints:

$$\begin{aligned} E_1 &= P_A + 2 \\ E_7 &= P_B - 4 \\ P_A &= P_B + 10 \\ \Longrightarrow E_1 &= E_7 + 16 \end{aligned}$$

In this example, it was possible to establish a global timescale for both events because there was an intersection between A's and B's receivers.

This technique for propagating timing information has many of the same benefits as in the single-hop case. A traditional solution to the multi-hop problem might be for nodes to re-broadcast a timing message that they have previously received. In contrast, our technique *never depends on the temporal relationship between a sender and a receiver.* As in the single-hop case, removing all the nondeterministic senders from the critical path reduces the error accumulated at each hop, as we will show in Section 5.3.

For simplicity, the example above spoke of sending "a pulse." Naturally, the phase and skew relationship between receivers in a neighborhood can be determined more precisely by using a larger number of pulses, as we described in Section 4. To take advantage of this information, our RBS prototype does not literally compare pulse reception times, as suggested by the constraints in the equations above. Instead, it performs a series of timebase conversions using the best-fit lines that relate logically-adjacent receivers to each other. Adopting the notation $E_i(R_j)$ to mean the time of event i according to receiver j's clock, we can state the multihop algorithm more precisely:

1. Receivers R_1 and R_7 observe events at times $E_1(R_1)$ and $E_7(R_7)$, respectively.

2. R_4 uses A's reference broadcasts to establish the best-fit line (as in Figure 4a) needed for converting clock values from R_1 to R_4. This line is used to convert $E_1(R_1)$ to $E_1(R_4)$.

3. R_4 similarly uses B's broadcasts to convert $E_1(R_4)$ to $E_1(R_7)$.

4. The time elapsed between the events is computed as $E_1(R_7) - E_7(R_7)$.

This algorithm is conceptually the same as the simpler version. However, each timebase conversion implicitly includes a correction for skew in all three nodes.

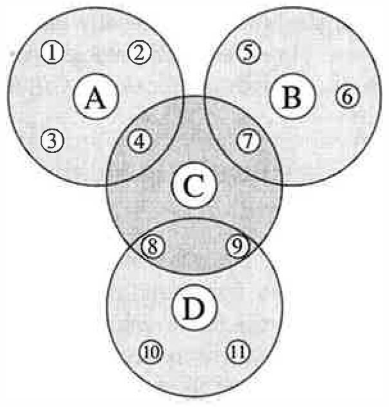
5.2 Time Routing in Multihop Networks

In Figure 6, there was a single "gateway" node that we designated for converting timestamps from one neighborhood's timebase to another. However, in practice, no such designation is necessary, or even desirable. It is straightforward to compute a "time route"—that is, dynamically determine a series of nodes through which time can be converted to reach a desired final timebase.

Consider the network topology in Figure 7a. As in Figure 6, the lettered nodes represent sync pulse senders; numbered nodes are receivers. (This distinction between senders and receiver roles is purely illustrative; in reality, a node can play both roles.) Each pulse sender creates a neighborhood of nodes that have known phase offsets by virtue of having received pulses from a common source. These relationships are visualized in Figure 7b, in which nodes with known clock relationships are joined by graph edges. A path through this graph represents a series of timebase conversions. For example, we can compare the time of $E_1(R_1)$ with $E_{10}(R_{10})$ by transforming $E_1(R_1) \rightarrow E_1(R_4) \rightarrow E_1(R_8) \rightarrow E_1(R_{10})$.

It is possible to find a series of conversions automatically, simply by performing a shortest-path search in a graph such as in Figure 7b. In addition, the weights of the graph edges can be used to represent the quality of the conversion—for example, the residual error (RMS) of the linear fit to the broadcast observations. A minimum-error conversion between any two nodes can be found using a weighted-shortest-path search such as Dijkstra's or Bellman-Ford.

Of course, such shortest-path algorithms do not scale to large networks due to their dependence on global information. Although there is precedent for such systems (e.g., the Internet's early link-state routing algorithms), a more localized approach is needed if the method is to scale. To this end, there is an interesting alternative: time conversion can be built into the extant packet forwarding mechanism. That is, nodes can perform successive time conversions on packets as they are forwarded from node to node—keeping the time with respect to the local clock at each hop. This technique, also suggested by Röemer [28], is attractive because it requires only local computation and communication. Instead of flooding clock conversion parameters globally, they can be distributed using a tightly scoped broadcast. In addition, the quality

a) 

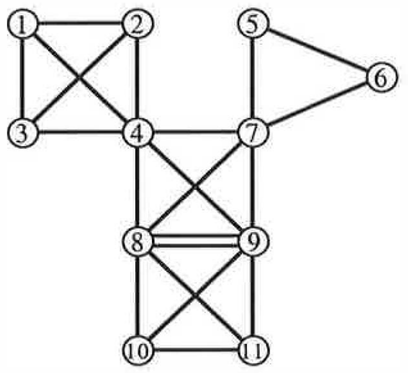
b) 

Figure 7: A more complex (3-hop) multihop network topology. *a)* The physical topology. Beacons (lettered nodes), whose transmit radii are indicated by the larger shaded areas, broadcast references to their neighbors (numbered nodes). *b)* The logical topology. Edges are drawn between nodes that have received a common broadcast, and therefore have enough information to relate their clocks to each other. Receivers 8 and 9 share two logical links because they have two receptions in common (from C and D).

of time synchronization across each link can be incorporated into the link metric used by the routing algorithm, ensuring that routing is not completely orthogonal to the quality of the time conversions available.

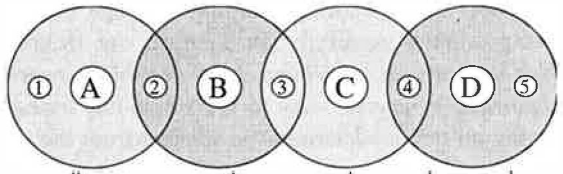
5.3 Measured Performance of Multihop RBS

In a multihop topology where a series of timestamp conversions are required, as described in the previous section, each conversion step can introduce synchronization error. We now consider the cumulative effects of such errors as the path length grows.

We saw in Section 3.2 that the synchronization error introduced by a reference broadcast is Gaussian. In addition, each of the per-hop contributions to the error are *independent* because the phase and skew estimates at each hop are based on a different set of broadcasts. Therefore, we expect that total path error—a sum of independent Gaussian variables—will also follow a Gaussian distribution. If the average per-hop error along the path is σ, then we expect[7] the average path error for an n-hop path to be $\sigma\sqrt{n}$. This bound is pleasing in that it grows slowly, and implies that we can synchronize nodes across many hops without significant decay in precision.

We tested the precision of multihop RBS using the IPAQ and 802.11-based testbed we described in Section 4.4, including the in-kernel packet timestamping discussed in Section 4.5. We configured 9 IPAQs in a linear topology, alternating between 5 (numbered) receivers and 4 (lettered) beacon emitters, as shown in Figure 8. Each receiver could hear only the nearest sender on its left and right. The test procedure was the same as we described in Section 4.4—in 300 trials, two IPAQs were commanded to raise a GPIO pin high at the same time. A logic analyzer reported the actual time differences. We tested a 1-hop, 2-hop, 3-hop and 4-hop conversion, by testing the rise-time difference between nodes 1–2, 1–3, 1-4, and 1–5, respectively.

The results, shown in Figure 8, generally seem to follow our $\sigma\sqrt{n}$ estimate. The average 4-hop error was $3.68 \pm 2.57\mu$sec, which is nearly $\sigma\sqrt{4}$, normalizing σ to be the average 1-hop error.



Hops	Mean Error	Std Dev	50%	95%	99%
1	1.85	1.28	1.60	4.51	5.85
2	2.73	1.91	2.41	6.49	7.74
3	2.73	2.42	2.22	6.93	9.92
4	3.68	2.57	3.18	8.26	10.70

Figure 8: *top)* The topology used to test cumulative error when RBS is used in a multihop network. *bottom)* Measured performance of multihop RBS, using kernel timestamping, on IPAQs using 11 MBit/sec 802.11. All units are μsec.

[7]From the variance's identity that $var(x+y) = var(x) + var(y)$; variance=$\sigma^2$.

5.4 Synchronization with External Timescales

So far, we have spoken entirely of creating local, internally consistent timescales. Although relative synchronization is sufficient for many applications, others require absolute time as measured by an external reference such as UTC. Reference timescales are typically distributed via governmentally sponsored radio systems such as the short-wave WWVB station [3] or the satellite-based Global Positioning System [16]. Commercially available receivers for these systems can synchronize computers by providing them with a "PPS" (pulse-per-second) signal at the beginning of each second. The seconds are numbered out of band, such as through a serial port.

RBS provides a natural and precise way to synchronize a network with such an external timescale. For example, consider a GPS radio receiver connected to one of the nodes in a multihop network such as the one in Figure 7. GPS time simply becomes another node in that network, attached via one edge to its host node. The PPS output of the receiver can be treated exactly as normal reference broadcasts are. That is, the host node timestamps each PPS pulse and compares its own timestamp with the pulse's true GPS time. The phase and skew of the host node's clock relative to GPS time can then be recovered using the algorithms described in Section 4. Other nodes can use GPS time by finding a multihop conversion route to it through the host node, using the algorithm we described in Section 5.2.

6 Application Experience

To date, our RBS implementation has been applied, both within and without our research group, in three distributed sensor network applications requiring high-precision time synchronization. Each uses RBS-synchronized clocks to precisely measure the arrival time of acoustic signals. Most audio codecs sample the channel at 48KHz, or $\approx 21\mu$sec per sample. As we have seen in previous sections, the precision nominally achieved by RBS is significantly better. Distributed acoustic sensors have thus been able to correlate their acoustic time-series down to individual samples. This has been useful in a number of contexts.

Our group has developed a implemented a centimeter-scale localization service for Berkeley Motes based on acoustic time-of-flight ranging [8]. A set of IPAQs set around the room first establish their own positions within a relatively coordinate system by ranging to one another. Each IPAQ emits an audible "chirp" which has an encoded pseudonoise sequence [9]. IPAQs run a matched filter over their incoming audio data to find the most likely audio sample indicating arrival of the chirp. 802.11-based RBS corrections are then applied to convert this into the equivalent sample number in the sender's timebase, allowing time of flight and therefore range to be computed. Once this startup phase is complete, our "acoustic Motes," specially equipped with small amplifiers and speakers, periodically emit a similar pseudonoise chirp; IPAQs in the region each compute their ranges to the Mote, and can localize it by trilaterating. In this case, RBS is used to synchronize all Motes in the system to each other. A special "MoteNIC"—a Mote physically attached to an IPAQ—provides translations between the Mote and IPAQ time domains.

Merrill *et al.* describe Sensoria Corporation's use of RBS in their implementation of a distributed, wireless embedded system capable of autonomous position location with accuracy on the order of 10cm [22]. This system was built under the auspices of the DARPA SHM program and has been field tested at Fort Leonard Wood, Missouri, in configurations of up to 20 nodes. Their application is also notable because it represents a third hardware and radio platform on which RBS has been successfully implemented. Sensoria's "WINS NG" node is based around the Hitachi SH4 processor running Linux 2.4. Each node has two low-power 56Kbit/sec hybrid TDMA/FHSS radios with an RS-232 serial interface to the host processor. As with our 802.11 implementation, the firmware of this radio was opaque, making schemes that rely on tricks at the MAC layer impossible.

Finally, blind beamforming on acoustic signals has been studied by Yao *et al.* for a number of years [35]. Their group had previously restricted their empirical studies to centralized systems, as they did not have a platform capable of synchronizing distributed audio sampling with sufficient precision. Recently, Wang *et al.* reported their first experience building a *distributed* beam-forming application on IPAQs using our RBS daemon [34].

7 Conclusions and Future Work

In this paper, we explored a form of time synchronization, *Reference-Broadcast Synchronization*, that provides more precise, flexible, and resource-efficient network time synchronization than traditional algorithms. The fundamental property of our design is that it *synchronizes a set of receivers with one another*, as opposed to traditional protocols in which senders synchronize with receivers. In addition, we have presented a novel multi-hop algorithm that allows this fundamental property to be maintained across broadcast domains.

In our scheme, nodes periodically send beacon messages to their neighbors using the network's physical-layer broadcast. Recipients use the message's arrival time as a point of reference for comparing their clocks. The message contains no explicit timestamp, nor is it important exactly when it is sent.

RBS has a number of properties that make it attractive. First, by using the broadcast channel to synchronize receivers with one another, the largest sources of nondeterministic latency are removed from the critical path. This results in significantly better-precision synchronization than algorithms that measure round-trip delay. The residual error is often a well-behaved distribution (e.g., Gaussian), allowing further improvement by sending multiple reference broadcasts. The extra information produces significantly better estimates of relative phase and frequency, and allows graceful degradation in the presence of outliers and lost packets.

The RBS clock skew estimate also supports extrapolation of *past* phase offsets. This enables nodes to synchronize *post-facto*, saving energy in applications that need synchronized time infrequently and unpredictably.

We presented a quantitative study that compared an RBS implementation to a carefully tuned installation of GPS-steered NTP. Both used IPAQ PDAs with 802.11 wireless Ethernet. We found that the average synchronization error of RBS was $6.29 \pm 6.45\mu$sec, 8 times better than that of NTP in a lightly loaded network. A heavily loaded network further degraded NTP's performance by a factor of 30 but had little effect on RBS. With kernel timestamping hints, RBS achieved an average error of $1.85 \pm 1.28\mu$sec, which we expect was limited by the IPAQ's 1μsec clock resolution.

Our multihop scheme allows locally coordinated timescales to be federated into a global timescale, across broadcast domains. Precision decays slowly—the average error for an n-hop network is proportional to $\sqrt{n}$. In our test of kernel-assisted RBS across a 4-hop topology, average synchronization error was $3.68 \pm 2.57\mu$sec. RBS-federated timescales may also be referenced to an external standard such as UTC if at least one node has access to a source of reference time.

We have implemented RBS on a variety of hardware platforms, where it has proven to be robust and reliable for both performance measurement and in support of real applications. However, there is still much work to be done. Some important scaling issues have not yet been explored, such as automatic, dynamic election of the set of nodes to act as beacon senders. If there are multiple beacon-senders in a single neighborhood, RBS can make use of redundant information to improve precision; however, it quickly reaches the point of diminishing return where the system would be better off conserving bandwidth instead. We would like understand just how much extra precision is afforded by this redundancy.

In addition, we would like to more fully quantify a number of performance questions: precision vs. both the bandwidth used for beacons, and their frequency; minimum time to acquire synchronization to within a given precision bound; post-facto synchronization precision vs. time elapsed from event to reference pulses; and precision with different data filtering algorithms. We would like to quantify the performance of RBS when used for external (UTC) synchronization over multiple hops, vs. using NTP in the same topology.

We are confident that these techniques are widely applicable, based on our experience with Berkeley Motes running TinyOS, Linux-based IPAQs, PCs, and Sensoria's WinsNG radios. Each has quirks that have taught us important lessons about RBS, so we would like further with experience with RBS with a wider range of hardware platforms, network interfaces, operating environments, and applications. Many of our collaborators in the sensor network community have research interests that require precise time synchronization: collaborative signal processing, acoustic ranging, object tracking, coordinated actuation, and so forth. We look forward to evaluating the utility of RBS in support of these applications.

Acknowledgments

This work was made possible with support from the DARPA NEST program (the "GALORE" project, grant F33615-01-C-1906). Additional support was provided through the University of California MICRO program (grant number 01-031) and matching funds from Intel Corporation. The authors are indebted to Sensoria Corporation for support and feedback. Members of the UCLA LECS laboratory, the anonymous reviewers, and OSDI shepherd Eric Brewer provided insightful comments that significantly improved our research.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38(4):393–422, March 2002.

[2] G. Asada, M. Dong, T.S. Lin, F. Newberg, G. Pottie, W.J. Kaiser, and H.O. Marcy. Wireless Integrated Network Sensors: Low Power Systems on a Chip. In *Proceedings of the European Solid State Circuits Conference*, 1998.

[3] R.E. Beehler. Time/frequency services of the U.S. National Bureau of Standards and some alternatives for future improvement. *Journal of Electronics and Telecommunications Engineers*, 27:389–402, Jan 1981.

[4] Alberto Cerpa, Jeremy Elson, Deborah Estrin, Lewis Girod, Michael Hamilton, and Jerry Zhao. Habitat monitoring: Application driver for wireless communications technology. In *Proceedings of the 2001 ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, April 2001. Available at http://www.isi.edu/scadds/papers/CostaRica-oct01-final.ps.

[5] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[6] Jeremy Elson and Deborah Estrin. Time synchronization for wireless sensor networks. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS-01)*. IEEE Computer Society, April 23–27 2001.

[7] Saurabh Ganeriwal, Ram Kumar, Sachin Adlakha, and Mani B. Srivastava. Network-wide Time Synchrinization in Sensor Networks. Technical report, University of California, Dept. of Electrical Engineering, 2002.

[8] Lewis Girod, Vladimir Bychkovskiy, Jeremy Elson, and Deborah Estrin. Locating tiny sensors in time and space: A case study. In *In Proceedings of the International Conference on Computer Design (ICCD 2002)*, Freiburg, Germany, September 2002. http://lecs.cs.ucla.edu/Publications.

[9] Lewis Girod and Deborah Estrin. Robust range estimation using acoustic and multimodal sensing. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001)*, March 2001.

[10] R. Gusell and S. Zatti. The accuracy of clock synchronization achieved by TEMPO in Berkeley UNIX 4.3 BSD. *IEEE Transactions on Software Engineering*, 15:847–853, 1989.

[11] Jason Hill and David Culler. A wireless embedded sensor architecture for system-level optimization. Technical report, U.C. Berkeley, 2001.

[12] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Cambridge, MA, USA, November 2000. ACM.

[13] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking*, pages 56–67, Boston, MA, August 2000. ACM Press.

[14] ISO/IEC. IEEE 802.11 Standard. IEEE Standard for Information Technology, ISO/IEC 8802-11:1999(E), 1999.

[15] J.M. Kahn, R.H. Katz, and K.S.J. Pister. Next century challenges: mobile networking for Smart Dust. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278, 1999.

[16] Elliott D. Kaplan, editor. *Understanding GPS: Principles and Applications*. Artech House, 1996.

[17] H. Kopetz and W. Schwabl. Global time in distributed real-time systems. Technical Report 15/89, Technische Universität Wien, 1989.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–65, 1978.

[19] Cheng Liao, Margaret Martonosi, and Douglas W. Clark. Experience with an adaptive globally-synchronizing clock algorithm. In *Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '99)*, pages 106–114, New York, June 1999.

[20] J. Mannermaa, K. Kalliomaki, T. Mansten, and S. Turunen. Timing performance of various GPS receivers. In *Proceedings of the 1999 Joint Meeting of the European Frequency and Time Forum and the IEEE International Frequency Control Symposium*, pages 287–290, April 1999.

[21] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In USENIX Association, editor, *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, pages 259–269, Berkeley, CA, USA, Winter 1993. USENIX.

[22] William Merrill, Lewis Girod, Jeremy Elson, Kathy Sohrabi, Fredric Newberg, and William Kaiser. Autonomous Position Location in Distributed, Embedded, Wireless Systems. In *Proceedings of IEEE CAS Workshop on Wireless Communications and Networking*, Pasadena, CA, September 2002. http://www.sensoria.com/publications.html.

[23] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 26(1):90–95, January 1983.

[24] David L. Mills. Internet Time Synchronization: The Network Time Protocol. In Zhonghua Yang and T. Anthony Marsland, editors, *Global States and Time in Distributed Systems*. IEEE Computer Society Press, 1994.

[25] David L. Mills. Precision synchronization of cmputer network clocks. *ACM Computer Comm. Review*, 24(2):28–43, April 1994.

[26] David L. Mills. Adaptive hybrid clock discipline algorithm for the network time protocol. *IEEE/ACM Transactions on Networking*, 6(5):505–514, October 1998.

[27] M. Mock, R. Frings, E. Nett, and S. Trikaliotis. Continuous clock synchronization in wireless real-time applications. In *The 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 125–133, Washington - Brussels - Tokyo, October 2000. IEEE.

[28] Kay Römer. Time synchronization in ad hoc networks. In *Proceedings of MobiHoc 2001*, Long Beach, CA, Oct 2001.

[29] I. Rubin. Message Delays in FDMA and TDMA Communication Channels. *IEEE Trans. Communin.*, COM27(5):769–777, May 1979.

[30] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *J-ACM*, 34(3):626–645, July 1987.

[31] Paulo Veríssimo and Luis Rodrigues. A posteriori agreement for fault-tolerant clock synchronization on broadcast networks. In Dhiraj K. Pradhan, editor, *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS '92)*, page 85, Boston, MA, July 1992. IEEE Computer Society Press.

[32] Paulo Veríssimo, Luis Rodrigues, and Antonio Casimiro. Cesiumspray: a precise and accurate global time service for large-scale systems. Tech. Rep. NAV-TR-97-0001, Universidade de Lisboa, 1997.

[33] John R. Vig. Introduction to Quartz Frequency Standards. Technical Report SLCET-TR-92-1, Army Research Laboratory, Electronics and Power Sources Directorate, October 1992. Available at http://www.ieee-uffc.org/freqcontrol/quartz/vig/vigtoc.htm.

[34] H. Wang, L. Yip, D. Maniezzo, J.C. Chen, R.E. Hudson, J.Elson, and K.Yao. A Wireless Time-Synchronized COTS Sensor Platform Part II–Applications to Beamforming. In *Proceedings of IEEE CAS Workshop on Wireless Communications and Networking*, Pasadena, CA, September 2002. http://lecs.cs.ucla.edu/Publications.

[35] K. Yao, R.E. Hudson, C.W. Reed, D. Chen, and F. Lorenzelli. Blind beamforming on a randomly distributed sensor array system. *IEEE Journal of Selected Areas in Communications*, 16(8):1555–1567, Oct 1998.

Supporting Time-Sensitive Applications on a Commodity OS

Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute, Portland
{ashvin,luca,jsnow,krasic,walpole}@cse.ogi.edu

Abstract

Commodity operating systems are increasingly being used for serving time-sensitive applications. These applications require low-latency response from the kernel and from other system-level services. In this paper, we explore various operating systems techniques needed to support time-sensitive applications and describe the design of our Time-Sensitive Linux (TSL) system. We show that the combination of a high-precision timing facility, a well-designed preemptible kernel and the use of appropriate scheduling techniques is the basis for a low-latency response system and such a system can have low overhead. We evaluate the behavior of realistic time-sensitive user- and kernel-level applications on our system and show that, in practice, it is possible to satisfy the constraints of time-sensitive applications in a commodity operating system without significantly compromising the performance of throughput-oriented applications.

1 Introduction

Multimedia applications, and soft real-time applications in general, are driven by real-world demands and are characterized by timing constraints that must be satisfied for correct operation; for this reason, we call these applications *time-sensitive*. Time-sensitive applications may require, for example, periodic execution with low jitter (e.g., soft modems [8]) or quick response to external events such as frame capture (e.g., video conferencing).

To support these time-sensitive applications, a general-purpose operating system (OS) must respect the application's timing constraints. To do so, resources must be allocated to the application at the appropriate times.

This work was partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs and by Intel.

This paper shows that there are three important requirements for achieving timely resource allocation: a high-precision timing facility, a well-designed preemptible kernel and the use of appropriate scheduling techniques. Each of these requirements have been addressed in the past with specific mechanisms, but unfortunately operating systems, such as Linux, often do not support or integrate these mechanisms.

This paper focuses on three specific techniques that can be integrated to satisfy the constraints of time-sensitive applications. First, we present *firm timers*, an efficient high-resolution timer mechanism. Firm timers incorporate the benefits of three types of timers, one-shot timers available on modern hardware [7], soft timers [2], and periodic timers to provide accurate timing with low overhead. Second, we use fine-grained kernel preemptibility to obtain a responsive kernel. Finally, we use both priority and reservation-based CPU scheduling mechanisms for supporting various types of time-sensitive applications. We have integrated these techniques in our extended version of the Linux kernel, which we call *Time-Sensitive Linux* (TSL).

Currently, commodity systems provide coarse-grained resource allocation with the goal of maximizing system throughput. Such a policy conflicts with the needs of time-sensitive applications which require more precise allocation. Thus, recently several approaches have been proposed to improve the timing response of a commodity system such as Linux [17, 22]. These approaches include improved kernel preemptibility and a more generic scheduling framework. However, since their focus is hard real-time, they do not evaluate the performance of non-real time applications.

In contrast, TSL focuses on integrating efficient support for time-sensitive applications in a commodity OS without significantly degrading the performance of traditional applications. Hence, one of the main contributions of this paper is to show through experimental evaluation that using the above techniques it is possible to provide good performance to time-sensitive applications as well as to throughput-oriented applications.

The rest of the paper is organized as follows. Section 2 investigates the factors that contribute to poor temporal response in commodity systems. Section 3 describes the techniques that we have used to implement TSL. Section 4 evaluates the behavior of several time-sensitive applications and presents overheads of TSL. Finally, in Section 5, we state our conclusions.

1.1 Related Work

The scheduling problem has been extensively studied by the real-time community [10, 19]. However, most of the scheduling analysis is based on an abstract mathematical model that ignores practical systems issues such as kernel non-preemptibility and interrupt processing overhead. Recently, many different real-time algorithms have been implemented in Linux and in other general-purpose kernels. For example, Linux/RK [17] implements Resource Reservations in the Linux kernel, and RED Linux [22] provides a generic real-time scheduling framework. These kernels tackle the practical systems issues mentioned above with techniques similar to the techniques presented in this paper. For example, kernel preemptibility is used by Timesys Linux and MontaVista Linux [12]. However, while these kernels work well for time-sensitive applications, their performance overhead on throughput-oriented applications is not clear.

The SMaRT [16] and Linux-SRT [6] kernels share our goal of supporting time-sensitive applications on commodity OSs. Before implementing SMaRT, Nieh, et al. [15] showed that tuning the time-sharing or the real-time priorities of applications required a great deal of experimentation in SVR4 Unix and often lead to unpredictable results. They showed empirically, similar to our priority-assignment protocol, that assigning a higher real-time priority to the X server improved the performance of video. Later, Nieh implemented the SMaRT real-time scheduler on Solaris that dynamically balances between the needs of different jobs by giving proportional shares to each job and a bias that improves the responsiveness of interactive and real-time jobs. This scheduler complements our approach of using priorities or proportion-period scheduling, as appropriate, for time-sensitive tasks.

Linux-SRT supports multiple real-time scheduling policies and implements a timing mechanism that is more accurate than standard Linux. However, it doesn't incorporate kernel preemption or discuss the issue of time-sensitive performance under heavy system load. Linux-SRT recognizes that the timing behavior of an application depends on shared system services such as the X server and thus modifies the X server to prioritize graphics rendering based on the scheduling parameters of tasks. We use a simpler priority-assignment protocol to achieve the same effect without requiring any modifications to the X server.

A different approach for providing real-time performance is used by systems such as RTLinux [4], which decrease timing unpredictability in the system by running Linux as a background process over a small real-time executive. This solution provides good real-time performance, but does not provide it to Linux applications. Also, native real-time threads use a different and less evolved application binary interface (ABI) as compared to the Linux interface and do not have access to Linux device drivers.

An accurate timing mechanism is crucial for supporting time-sensitive applications. Thus most of the existing real-time kernels or real-time extensions to Linux provide high resolution timers. The high resolution timers concept was proposed by RT-Mach [18] and has subsequently been used by several other systems [17]. In a general-purpose operating system, the overhead of such timers can affect the performance of throughput-oriented applications. This overhead is caused by the increased number of interrupts generated by the fine-grained timing mechanism and can be mitigated by the soft-timer mechanism [2]. Thus, our firm-timer implementation uses soft timers.

Finally, the Nemesis operating system [9] is designed for multimedia and other time-sensitive applications. However, its structure and API is very different from the standard programming environment provided by operating systems such as Linux. Our goal is to minimize changes to the programming environment to encourage the use of time-sensitive applications in a commodity environment.

2 Time-Sensitive Requirements

To satisfy a time-sensitive application's temporal constraints, resources must be allocated at "appropriate" times. These appropriate times are determined by events of interest to the application, such as readiness of a video frame for display. In response to these events, the application is scheduled or activated. Hence, we view the time-line of the application as a sequence of such *events* and the corresponding *activations*. For example, Fig-

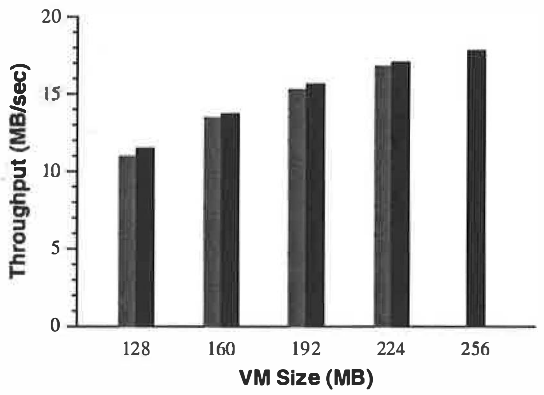


Figure 2: **Balloon Performance.** Throughput of single Linux VM running dbench with 40 clients. The black bars plot the performance when the VM is configured with main memory sizes ranging from 128 MB to 256 MB. The gray bars plot the performance of the same VM configured with 256 MB, ballooned down to the specified size.

or removed from a VM in order to rapidly adjust its physical memory size.

To demonstrate the effectiveness of ballooning, we used the synthetic dbench benchmark [28] to simulate fileserver performance under load from 40 clients. This workload benefits significantly from additional memory, since a larger buffer cache can absorb more disk traffic. For this experiment, ESX Server was running on a dual-processor Dell Precision 420, configured to execute one VM running Red Hat Linux 7.2 on a single 800 MHz Pentium III CPU.

Figure 2 presents dbench throughput as a function of VM size, using the average of three consecutive runs for each data point. The ballooned VM tracks non-ballooned performance closely, with an observed overhead ranging from 4.4% at 128 MB (128 MB balloon) down to 1.4% at 224 MB (32 MB balloon). This overhead is primarily due to guest OS data structures that are sized based on the amount of "physical" memory; the Linux kernel uses more space in a 256 MB system than in a 128 MB system. Thus, a 256 MB VM ballooned down to 128 MB has slightly less free space than a VM configured with exactly 128 MB.

Despite its advantages, ballooning does have limitations. The balloon driver may be uninstalled, disabled explicitly, unavailable while a guest OS is booting, or temporarily unable to reclaim memory quickly enough to satisfy current system demands. Also, upper bounds on reasonable balloon sizes may be imposed by various guest OS limitations.

3.3 Demand Paging

ESX Server preferentially uses ballooning to reclaim memory, treating it as a common-case optimization. When ballooning is not possible or insufficient, the system falls back to a paging mechanism. Memory is reclaimed by paging out to an ESX Server swap area on disk, without any guest involvement.

The ESX Server swap daemon receives information about target swap levels for each VM from a higher-level policy module. It manages the selection of candidate pages and coordinates asynchronous page outs to a swap area on disk. Conventional optimizations are used to maintain free slots and cluster disk writes.

A randomized page replacement policy is used to prevent the types of pathological interference with native guest OS memory management algorithms described in Section 3.1. This choice was also guided by the expectation that paging will be a fairly uncommon operation. Nevertheless, we are investigating more sophisticated page replacement algorithms, as well policies that may be customized on a per-VM basis.

4 Sharing Memory

Server consolidation presents numerous opportunities for sharing memory between virtual machines. For example, several VMs may be running instances of the same guest OS, have the same applications or components loaded, or contain common data. ESX Server exploits these sharing opportunities, so that server workloads running in VMs on a single machine often consume less memory than they would running on separate physical machines. As a result, higher levels of overcommitment can be supported efficiently.

4.1 Transparent Page Sharing

Disco [3] introduced *transparent page sharing* as a method for eliminating redundant copies of pages, such as code or read-only data, across virtual machines. Once copies are identified, multiple guest "physical" pages are mapped to the same machine page, and marked copy-on-write. Writing to a shared page causes a fault that generates a private copy.

Unfortunately, Disco required several guest OS modifications to identify redundant copies as they were created. For example, the bcopy() routine was hooked to valuable is known only by the guest operating system within each VM. Although there is no shortage of clever page replacement algorithms [26], this is actually the crux of the problem. A sophisticated meta-level policy is likely to introduce performance anomalies due to unintended interactions with native memory management policies in guest operating systems. This situation is exacerbated by diverse and often undocumented guest OS policies [1], which may vary across OS versions and may even depend on performance hints from applications [4].

The fact that paging is transparent to the guest OS can also result in a *double paging* problem, even when the meta-level policy is able to select the same page that the native guest OS policy would choose [9, 20]. Suppose the meta-level policy selects a page to reclaim and pages it out. If the guest OS is under memory pressure, it may choose the very same page to write to its own virtual paging device. This will cause the page contents to be faulted in from the system paging device, only to be immediately written out to the virtual paging device.

3.2 Ballooning

Ideally, a VM from which memory has been reclaimed should perform as if it had been configured with less memory. ESX Server uses a *ballooning* technique to achieve such predictable performance by coaxing the guest OS into cooperating with it when possible. This process is depicted in Figure 1.

A small *balloon* module is loaded into the guest OS as a pseudo-device driver or kernel service. It has no external interface within the guest, and communicates with ESX Server via a private channel. When the server wants to reclaim memory, it instructs the driver to "inflate" by allocating pinned physical pages within the VM, using appropriate native interfaces. Similarly, the server may "deflate" the balloon by instructing it to deallocate previously-allocated pages.

Inflating the balloon increases memory pressure in the guest OS, causing it to invoke its own native memory management algorithms. When memory is plentiful, the guest OS will return memory from its free list. When memory is scarce, it must reclaim space to satisfy the driver allocation request. The guest OS decides which particular pages to reclaim and, if necessary, pages them out to its own virtual disk. The balloon driver communicates the physical page number for each allocated page to ESX Server, which may then reclaim the corresponding machine page. Deflating the balloon frees up memory for general use within the guest OS.

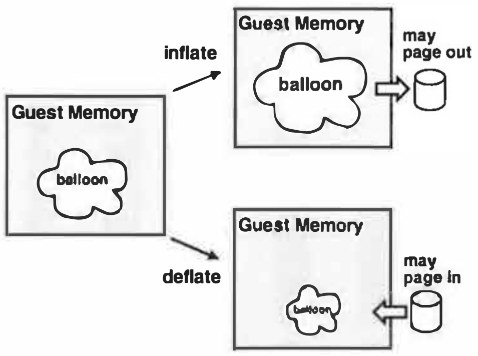


Figure 1: **Ballooning.** ESX Server controls a *balloon* module running within the guest, directing it to allocate guest pages and pin them in "physical" memory. The machine pages backing this memory can then be reclaimed by ESX Server. Inflating the balloon increases memory pressure, forcing the guest OS to invoke its own memory management algorithms. The guest OS may page out to its virtual disk when memory is scarce. Deflating the balloon decreases pressure, freeing guest memory.

Although a guest OS should not touch any physical memory it allocates to a driver, ESX Server does not depend on this property for correctness. When a guest PPN is ballooned, the system annotates its *pmap* entry and deallocates the associated MPN. Any subsequent attempt to access the PPN will generate a fault that is handled by the server; this situation is rare, and most likely the result of complete guest failure, such as a reboot or crash. The server effectively "pops" the balloon, so that the next interaction with (any instance of) the guest driver will first reset its state. The fault is then handled by allocating a new MPN to back the PPN, just as if the page was touched for the first time.[2]

Our balloon drivers for the Linux, FreeBSD, and Windows operating systems poll the server once per second to obtain a target balloon size, and they limit their allocation rates adaptively to avoid stressing the guest OS. Standard kernel interfaces are used to allocate physical pages, such as `get_free_page()` in Linux, and `MmAllocatePagesForMdl()` or `MmProbeAndLockPages()` in Windows.

Future guest OS support for hot-pluggable memory cards would enable an additional form of coarse-grained ballooning. Virtual memory cards could be inserted into

[2]ESX Server zeroes the contents of newly-allocated machine pages to avoid leaking information between VMs. Allocation also respects cache coloring by the guest OS; when possible, distinct PPN colors are mapped to distinct MPN colors.

This paper introduces several novel mechanisms and policies that ESX Server 1.5 [29] uses to manage memory. High-level resource management policies compute a target memory allocation for each VM based on specified parameters and system load. These allocations are achieved by invoking lower-level mechanisms to reclaim memory from virtual machines. In addition, a background activity exploits opportunities to share identical pages between VMs, reducing overall memory pressure on the system.

In the following sections, we present the key aspects of memory resource management using a bottom-up approach, describing low-level mechanisms before discussing the high-level algorithms and policies that coordinate them. Section 2 describes low-level memory virtualization. Section 3 discusses mechanisms for reclaiming memory to support dynamic resizing of virtual machines. A general technique for conserving memory by sharing identical pages between VMs is presented in Section 4. Section 5 discusses the integration of working-set estimates into a proportional-share allocation algorithm. Section 6 describes the high-level allocation policy that coordinates these techniques. Section 7 presents a remapping optimization that reduces I/O copying overheads in large-memory systems. Section 8 examines related work. Finally, we summarize our conclusions and highlight opportunities for future work in Section 9.

2 Memory Virtualization

A *guest* operating system that executes within a virtual machine expects a zero-based physical address space, as provided by real hardware. ESX Server gives each VM this illusion, virtualizing physical memory by adding an extra level of address translation. Borrowing terminology from Disco [3], a *machine address* refers to actual hardware memory, while a *physical address* is a software abstraction used to provide the illusion of hardware memory to a virtual machine. We will often use "physical" in quotes to highlight this deviation from its usual meaning.

ESX Server maintains a *pmap* data structure for each VM to translate "physical" page numbers (PPNs) to machine page numbers (MPNs). VM instructions that manipulate guest OS page tables or TLB contents are intercepted, preventing updates to actual MMU state. Separate *shadow page tables*, which contain virtual-to-machine page mappings, are maintained for use by the processor and are kept consistent with the physical-to-machine mappings in the *pmap*.[1] This approach permits ordinary memory references to execute without additional overhead, since the hardware TLB will cache direct virtual-to-machine address translations read from the shadow page table.

The extra level of indirection in the memory system is extremely powerful. The server can remap a "physical" page by changing its PPN-to-MPN mapping, in a manner that is completely transparent to the VM. The server may also monitor or interpose on guest memory accesses.

3 Reclamation Mechanisms

ESX Server supports overcommitment of memory to facilitate a higher degree of server consolidation than would be possible with simple static partitioning. Overcommitment means that the total size configured for all running virtual machines exceeds the total amount of actual machine memory. The system manages the allocation of memory to VMs automatically based on configuration parameters and system load.

Each virtual machine is given the illusion of having a fixed amount of physical memory. This *max size* is a configuration parameter that represents the maximum amount of machine memory it can be allocated. Since commodity operating systems do not yet support dynamic changes to physical memory sizes, this size remains constant after booting a guest OS. A VM will be allocated its maximum size when memory is not overcommitted.

3.1 Page Replacement Issues

When memory is overcommitted, ESX Server must employ some mechanism to reclaim space from one or more virtual machines. The standard approach used by earlier virtual machine systems is to introduce another level of paging [9, 20], moving some VM "physical" pages to a swap area on disk. Unfortunately, an extra level of paging requires a meta-level page replacement policy: the virtual machine system must choose not only the VM from which to revoke memory, but also which of its particular pages to reclaim.

In general, a meta-level page replacement policy must make relatively uninformed resource management decisions. The best information about which pages are least

[1]The IA-32 architecture has hardware mechanisms that walk in-memory page tables and reload the TLB [13].

Memory Resource Management in VMware ESX Server

Carl A. Waldspurger

VMware, Inc.
Palo Alto, CA 94304 USA
carl@vmware.com

Abstract

VMware ESX Server is a thin software layer designed to multiplex hardware resources efficiently among virtual machines running unmodified commodity operating systems. This paper introduces several novel ESX Server mechanisms and policies for managing memory. A *ballooning* technique reclaims the pages considered least valuable by the operating system running in a virtual machine. An *idle memory tax* achieves efficient memory utilization while maintaining performance isolation guarantees. *Content-based page sharing* and *hot I/O page remapping* exploit transparent page remapping to eliminate redundancy and reduce copying overheads. These techniques are combined to efficiently support virtual machine workloads that overcommit memory.

1 Introduction

Recent industry trends, such as server consolidation and the proliferation of inexpensive shared-memory multiprocessors, have fueled a resurgence of interest in server virtualization techniques. Virtual machines are particularly attractive for server virtualization. Each *virtual machine (VM)* is given the illusion of being a dedicated physical machine that is fully protected and isolated from other virtual machines. Virtual machines are also convenient abstractions of server workloads, since they cleanly encapsulate the entire state of a running system, including both user-level applications and kernel-mode operating system services.

In many computing environments, individual servers are underutilized, allowing them to be consolidated as virtual machines on a single physical server with little or no performance penalty. Similarly, many small servers can be consolidated onto fewer larger machines to simplify management and reduce costs. Ideally, system administrators should be able to flexibly overcommit memory, processor, and other resources in order to reap the benefits of statistical multiplexing, while still providing resource guarantees to VMs of varying importance.

Virtual machines have been used for decades to allow multiple copies of potentially different operating systems to run concurrently on a single hardware platform [8]. A *virtual machine monitor (VMM)* is a software layer that virtualizes hardware resources, exporting a virtual hardware interface that reflects the underlying machine architecture. For example, the influential VM/370 virtual machine system [6] supported multiple concurrent virtual machines, each of which believed it was running natively on the IBM System/370 hardware architecture [10]. More recent research, exemplified by Disco [3, 9], has focused on using virtual machines to provide scalability and fault containment for commodity operating systems running on large-scale shared-memory multiprocessors.

VMware *ESX Server* is a thin software layer designed to multiplex hardware resources efficiently among virtual machines. The current system virtualizes the Intel IA-32 architecture [13]. It is in production use on servers running multiple instances of unmodified operating systems such as Microsoft Windows 2000 Advanced Server and Red Hat Linux 7.2. The design of ESX Server differs significantly from VMware Workstation, which uses a *hosted* virtual machine architecture [23] that takes advantage of a pre-existing operating system for portable I/O device support. For example, a Linux-hosted VMM intercepts attempts by a VM to read sectors from its virtual disk, and issues a `read()` system call to the underlying Linux host OS to retrieve the corresponding data. In contrast, ESX Server manages system hardware directly, providing significantly higher I/O performance and complete control over resource management.

The need to run existing operating systems without modification presented a number of interesting challenges. Unlike IBM's mainframe division, we were unable to influence the design of the guest operating systems running within virtual machines. Even the Disco prototypes [3, 9], designed to run unmodified operating systems, resorted to minor modifications in the IRIX kernel sources.

[3] T. P. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, pages 67–99, 1991.

[4] Michael Barabanov and Victor Yodaiken. Real-time Linux. *Linux Journal*, March 1996.

[5] Randy Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *CACM*, 31(10):1220–1227, October 1988.

[6] Stephan Childs and David Ingram. The Linux-SRT integrated multimedia system: Bringing QoS to the desktop. In *Real-Time Technology and Applications Symposium*, May 2001.

[7] Intel Corporation, editor. *Pentium Pro Family Developer's Manual*, chapter 7.4.15. Intel, December 1995.

[8] Michael B. Jones and Stefan Saroiu. Predictability requirements of a softmodem. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Cambridge, MA, June 2001.

[9] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[10] C. L. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, Jan 1973.

[11] Robert Love. The Linux kernel preemption project. `http://kpreempt.sf.net`.

[12] Montavista Software - Powering the embedded revolution. `http://www.mvista.com`.

[13] Andrew Morton. Linux scheduling latency. `http://www.zip.com.au/~akpm/linux/schedlat.html`.

[14] Mplayer - Movie player for linux. `http://www.mplayerhq.hu`.

[15] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. In *NOSSDAV*, November 1993.

[16] Jason Nieh and Monica Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Symposium on Operating Systems Principles*, October 1997.

[17] Shui Oikawa and Raj Rajkumar. Linux/RK: A portable resource kernel in Linux. In *IEEE Real-Time Systems Symposium*, December 1998. Work-In-Progress Session.

[18] S. Savage and H. Tokuda. RT-Mach timers: Exporting time to the user. In *USENIX 3rd Mach Symposium*, April 1993.

[19] Lui Sha, Raghunathan Rajkumar, and John Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1184, September 1990.

[20] David Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Operating Systems Design and Implementation*, February 1999.

[21] The X window system. `http://www.x.org`.

[22] Yu-Chung and Kwei-Jay Lin. Enhancing the real-time capability of the Linux kernel. In *IEEE Real Time Computing Systems and Applications*, October 1998.

cost of checking if some soft timer has expired. Note that we described the components of these costs in the beginning of this section. The total cost of firing firm timers is $C_cN_c + C_hN_h + C_sN_s$. If pure hard timers are used then the cost is C_hN_t. Hence, firm timers reduce overhead if $C_cN_c + C_hN_h + C_sN_s < C_hN_t$. After substituting $N_t = N_h + N_s$, this equation simplifies to $N_s/N_c > C_c/(C_h - C_s)$.

Hence, when the ratio of the number of the soft timers that fire to the number of soft timer checks is sufficiently large (i.e., it is larger than $C_c/(C_h - C_s)$), then firm timers are effective in reducing the overhead of one-shot timers. From our experiments, we have extrapolated that $C_h = 8$ μs, $C_s = 1$ μs, and $C_c = 0.15$ μs, hence the firm timers mechanism becomes effective when $N_s/N_c > 0.15/(8-1) = 0.021$, or when more than 2.1% of the soft timer checks result in the firing of a soft timer.

Note that the number of checks N_c depends on the amount of interrupts and system calls that occur in the machine, whereas the number of soft timers that fire N_s depends on how the checks and the timers' deadlines are distributed in time and the overshoot value. Aron and Druschel's original work on soft timers [2] studied these distributions for a number of workloads. Their results show that for many workloads the distributions are such that checks often occur close to deadlines (thus increasing N_s/N_c), although how close is very workload dependent. Firm timers have the benefit of assuring low timer latency even for workloads with poor distributions, yet retaining the performance benefits of soft timers when the workload permits.

Note that soft timer checks are normally placed at kernel exit points where kernel critical sections end and where the scheduler function can be invoked. The use of a preemptible kernel design in TSL reduces the granularity of non-preemptible sections in the kernel and potentially allows more frequent soft timer checks at the end of spinlocks and hence can provide better timing accuracy. The key issue here is the overhead of this approach, which depends on the ratio N_s/N_c, i.e., whether sufficient additional soft timers fire as a result of the additional soft checks. While our current firm timer implementation does not check for timers at the end of each spinlock, we plan to evaluate this approach in the future.

5 Conclusions

This paper describes the design and implementation of a Time-Sensitive Linux (TSL) system that can support applications requiring fine-grained resource allocation and low-latency response. The three key techniques that we have investigated in the context of TSL are firm timers for accurate timing, fine-grained kernel preemptibility for improving kernel responsiveness and proportion-period scheduling for providing precise allocations to tasks. Our experiments show that integrating these techniques helps provide allocations to time-sensitive tasks with a variation of less than 400 us even under heavy CPU, disk and file system load. We show that the overhead of TSL on throughput-oriented applications is low and thus such a system is truly a general-purpose system since it can be used effectively for both time-sensitive and traditional interactive and long-running batch applications.

Although the first results presented in this paper are promising, TSL still need further investigation, since there are open issues related to interrupt service times, fine-grained accounting of time, latencies caused by network processing, and firm timers performance. For firm timers in particular, we are interested in investigating whether real workloads commonly lead to the $N_s/N_c > K$ condition under which firm timers are effective.

Acknowledgments

Andrew Black, Wu-chi Feng and Wu-chang Feng provided many useful suggestions on the initial draft of the paper. We would like to thank the reviewers, especially our shepherd, Timothy Roscoe, for taking the time and suggesting numerous improvements to the paper.

References

[1] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. A measurement-based analysis of the real-time performance of the Linux kernel. In *Real Time Technology and Applications Symposium (RTAS)*, September 2002.

[2] Mohit Aron and Peter Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, August 2000.

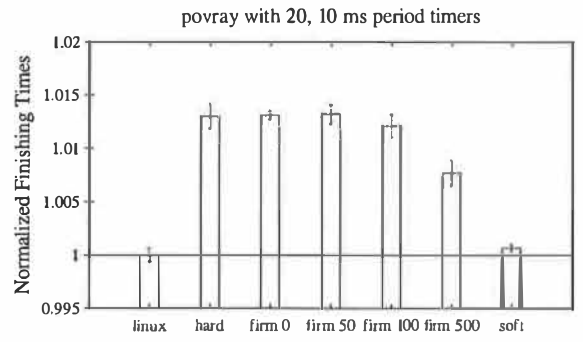


Figure 6: Overhead of firm timers in TSL as compared to standard Linux with 20 timer processes.

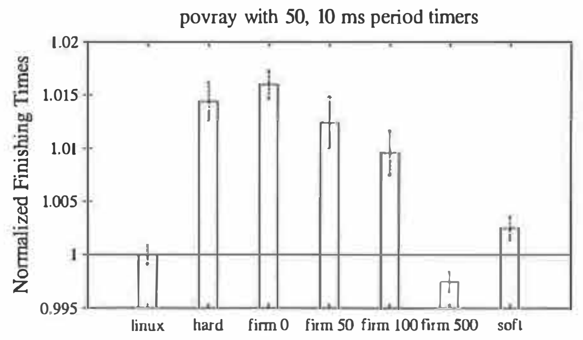


Figure 7: Overhead of firm timers in TSL as compared to standard Linux with 50 timer processes.

nounced in this case. The reason is that with increasing number of timers, timers are more likely to fire due to soft timers than the more expensive hardware APIC timer.

Interestingly, in Figure 7, the `povray` program completes faster on TSL with 500 μs firm-timer overshoot than on a standard Linux kernel. The reason for this apparent discrepancy is that the standard Linux scheduler does not scale well with large numbers of processes. On Linux, all 50 processes are woken at the same time (at the periodic timer interrupt boundary) and thus Linux has to schedule 50 processes at the same time. In comparison, on a firm timers kernel the 50 timers have precise 10 ms expiration times and are not synchronized. Hence, the scheduler generally has to schedule one or a small number of processes when a firm timer expires. In this case, the overhead of the scheduler on standard Linux dominates the overhead of the firm timers mechanism in TSL.

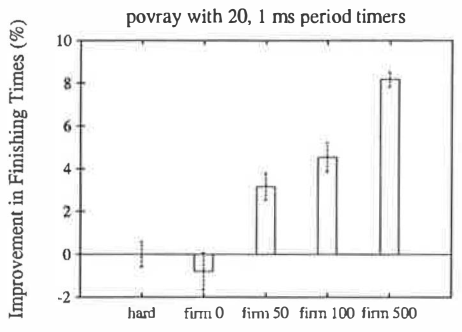


Figure 8: Comparison between hard and firm timers with different overshoot values on TSL.

Overhead at High Frequencies We also performed the same experiment but with periodic processes running at higher frequencies to simulate time-sensitive applications that have periodic timing requirements tighter than standard Linux can support. Figure 8 shows the improvement in time to render the image when 20 periodic processes are run with a period of 1 ms. We do not compare these results with Linux because Linux does not support 1 ms timer accuracy. Similarly, pure soft timers are not shown in this figure because they do not guarantee that each timer fires every 1 ms. This figure shows the improvement in finishing time of `povray` with firm timers with different overshoot values compared to hard timers. The benefit of the firm timers mechanism for improving throughput becomes more obvious with increasing overshoot when the process periods are made shorter. For example, there is an 8% improvement with a 500 μs overshoot value while the corresponding improvement in Figures 6 and 7 is 0.5% and 1.6%.

Discussion The previous experiments show that pure hard timers have lower overhead in some cases and firm timers have lower overhead in other cases. This result can be explained by the fact that there is a cost associated with checking whether a soft timer has expired. Thus, the soft timers mechanism is effective in reducing overhead when enough of these checks result in the firing of a soft timer. Otherwise the firm-timer overhead as compared to pure hard timers will be higher.

The previous behavior can be explained as follows. Let N_t be the total number of timers that must fire in a given interval of time, N_h the number of hard timers that fire, N_s the number of soft timers that fire (hence, $N_t = N_h + N_s$) and N_c the number of checks for soft timers expirations. Let C_h be the cost for firing a hard timer, C_s be the cost of firing a soft timer, and C_c be the timers were disabled in this experiment because we did not want to measure the cost of checking for soft timers.

The memory test under TSL has an overhead of 0.42 ± 0.18 percent while the fork test has an overhead of 0.53 ± 0.06 percent. The file system test did not have a significant overhead (in terms of confidence intervals). These tests indicate that the overhead of checking for preemption points in TSL compared to standard Linux is very low.

4.3.2 Firm Timers

The second cost in TSL is associated with executing firm timers. Firm timers provide an accurate timing mechanism which allows high frequency timer programming. However, increasing the timer frequency can increase system overhead because each timer event can cause a one-shot timer interrupt, which results in cache pollution. To mitigate this overhead, our firm timers implementation combines one-shot (or hard) timers and soft timers. In this section, we present experiments to highlight the advantages of firm timers as compared to hard timers and show that the overhead of firm timers on throughput-based applications is small even when firm timers are used heavily.

The cost of firm timers can be broken into three parts: 1) costs associated with hard timers exclusively, 2) costs that hard and soft timers have in common, and 3) costs associated with soft timers exclusively. The first cost occurs due to interrupt handling and the resulting cache pollution. The second cost lies in manipulating and dispatching timers from the timer queue and executing preemption for an expired timer thread. The third cost is in checking for soft timers. Note that the cost of executing preemption is present in both cases and thus the experiments presented below account for this cost when firm timers are used. Based on this breakup, it should be obvious that the soft timing component of firm timers will have lower overhead than hard timers if the cost for checking for timer expiry is less than the additional cost of interrupt handling in the pure hard timer case. This relation is derived in more detail below.

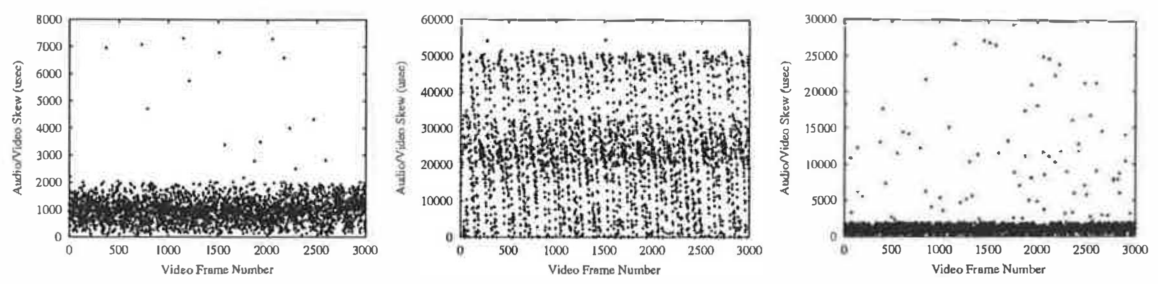
We will first compare the performance overhead of firm timers under TSL versus standard timers in Linux. This comparison is performed using multiple applications that each require 10 ms periodic timing. This case is favorable to Linux because the periodic timing mechanism in Linux synchronizes all the timers and generates one timer interrupt for all the threads, although at the expense of timer latency. In contrast, firm timers provide accurate timing but can generate multiple interrupts within each 10 ms period. Then we will evaluate the performance of firm timers for applications that require tighter timing than standard Linux can support.

In the following experiments, we measure the execution time of a throughput-oriented application when one or more time-sensitive processes are run in the background to stress the firm timers mechanism. We implement a time-sensitive process as a simple periodic task that wakes up on a signal generated by a firm timer (using the `setitimer()` system call), measures the current time and then immediately goes to sleep each period. In the rest of this section, we refer to this task as a timer process. For the throughput application, we selected `povray`, a ray-tracing application and used it to render a standard benchmark image called `skyvase`. We chose `povray` because it is a compute intensive job with a large memory footprint. Thus our experiments account for the effects of cache pollution due to the fine-grained timer interrupts. The performance overhead of firm timers is defined as the ratio of the time needed by `povray` to render the image in TSL versus the time needed to render the same image in standard Linux.

Comparison with Standard Linux We first compare the performance overhead of firm timers on TSL with standard timers running on Linux. To do so, we run timer processes with a 10 ms period because this period is supported by the tick interrupt in Linux. As explained above, we expect additional overhead in the firm timers case because, unlike with the periodic timers in Linux, the expiration times of the firm timers are not aligned. To stress the firm timers mechanism and clearly establish the performance difference, we ran two experiments with a large number of 20 and 50 timers processes.

Figure 6 shows the performance overhead of firm timers as compared to standard Linux timers when 20 timer processes are running simultaneously. This figure shows the overhead of TSL with hard timers, firm timers with different overshoot values (0 μs, 50 μs, 100 μs, 500 μs) and pure soft timers. Each experiment was run 8 times and the 95% confidence intervals are shown at the top of each bar. The figure shows that pure soft timers have an insignificant overhead compared to standard Linux while hard and firm timers have a 1.5 percent overhead. In this case, increasing the overshoot parameter of firm timers produces only a small improvement.

Figure 7 shows the results of the same experiment but with 50 timers. Once again soft timers have an insignificant overhead. In addition, the decrease in overhead of firm timers with increasing overshoot is more pro-



(a) User-level CPU load (b) Kernel CPU load (c) File-system load

Figure 5: Audio/Video Skew on Linux-SRT with user-level CPU load, kernel CPU load and file-system load.

	No Load		File System Load	
	Max Proportion Deviation	Max Period Deviation	Max Proportion Deviation	Max Period Deviation
Thread 1 Proportion: 40%, 3276.8 μs Period: 8192 μs	0.3% ($\simeq$25 μs)	5 μs	6% ($\simeq$490 μs)	534 μs
Thread 2 Proportion: 20%, 102.4 μs Period: 512 μs	0.7% ($\simeq$3 μs)	10 μs	4% ($\simeq$20 μs)	97 μs

Table 1: Deviation in proportion and period when two processes are run under the proportion-period scheduler in TSL.

the idea that the feedback scheduler should be informed about scheduling "errors" as much as possible, which was especially important because, before TSL, these errors were significantly larger. In the future, we plan to investigate improving the performance of proportion-period scheduling in the presence of heavy file system load by explicitly scheduling interrupt processing.

4.3 System Overhead

In this section, we focus on the performance overheads of TSL. There are two main sources of overhead in TSL as compared to standard Linux: 1) the cost of executing code at the newly inserted preemption points, and 2) the cost of executing firm timers.

4.3.1 Checking for Preemption

At each preemption point, there is a cost associated with checking for preemption, and then if scheduling is needed, there is a cost for executing preemption. We do not explicitly measure the second cost because it depends on the workload. However, one instance where we expect that more preemption will occur in TSL is when firm timers are used, since firm timers can cause preemption at a finer granularity. Hence, we discuss this cost later when we present the overhead of firm timers.

We measured the cost of the additional preemption checks in TSL by running a set of benchmarks that are known to stress preemption latency in Linux [1]. In particular, we ran three separate tests, a memory access test, a fork test and a file-system access test. Note that these tests are designed to stress preemption checks and thus measure their worst-case overhead. We expect that these checks will have a smaller impact on real applications. The memory test sequentially accesses a large integer array of 128 MB and thus produces several page faults in succession. The fork test creates 512 processes as quickly as possible. The file-system test repeatedly copies data from a 2 MB user buffer to a file that is 8 MB long and flushes the buffer cache. By running these tests, we expect to hit the various additional preemption checks that exist in TSL as compared to Linux. We measured the ratio of the completion times of these tests under TSL and under Linux in single user mode. Since no other process is running, these tests do not cause additional preemption and thus we are able to evaluate the cost of checking the additional preemption points. Firm

Comparison with Linux-SRT We also performed the previous experiments on Linux-SRT [6], which improves support for real-time applications by providing finer-grained timing than standard Linux, a reservation scheduler, and a modification to the X server to prioritize graphics rendering based on the scheduling parameters of tasks. Figure 5 shows the results of the experiments. For the non-kernel CPU load, the audio/video skew in Linux-SRT, when the X server was run at real-time priority, was generally less than 2 ms although the worst case latency is 7 ms. In this test, latency is dominated by timer latency, which is 1 ms on Linux-SRT. With the kernel CPU load test, the worst case latency was 60 ms, while the file-system test produced worst case latencies of 30 ms. These latter tests stress kernel preemption while Linux-SRT is a non-preemptive kernel. These results show that real-time scheduling and more precise timers are insufficient for time-sensitive applications, and that a responsive kernel is also required.

4.2.2 Proportion-Period Scheduler

As explained earlier, the proportion-period scheduler provides temporal protection when multiple time-sensitive tasks are run together. Our original motivation for implementing the TSL system was to provide an accurate implementation of a proportion-period scheduler. We use this scheduler to provide a fine-grained reservation mechanism for a higher-level feedback-based real-rate scheduler [20]. The problem with standard proportion-period scheduling is that it is difficult to correctly estimate a thread's proportion and period requirements in a general-purpose environment. To address this problem, the real-rate scheduler uses an application-specific progress rate metric in time-sensitive tasks to automatically assign correct allocations to such tasks. For example, the progress of a producer or consumer of a bounded buffer can be inferred by measuring the fill-level of the bounded buffer. If the buffer is full, the consumer is falling behind and needs more resources while the producer needs to be slowed down.

The accuracy of allocating resources using a feedback controller depends, among other factors, on the accuracy of actuating proportions. There are three sources of inaccuracy in our proportion-period scheduler implementation on standard Linux: 1) the period boundaries are quantized to multiples of the timer resolution or 10 ms, 2) the policing of proportions is also limited to the same value because timers have to be used to implement policing, and 3) heavy loads cause long non-preemptible paths and thus large jitter in period boundaries and proportion policing. These inaccuracies introduce noise in the system that can cause large allocation fluctuations even when the input progress signal can be captured perfectly and the controller is well-tuned.

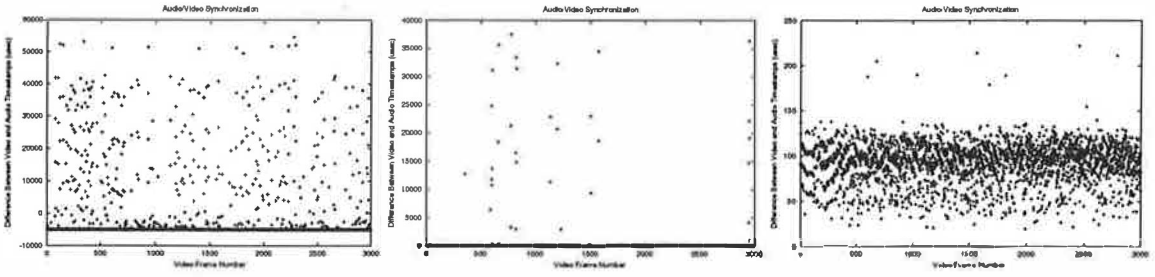
The proportion-period scheduler implementation on TSL uses firm-timers for implementing period boundaries and proportion policing. To evaluate the accuracy of this scheduler when multiple time-sensitive applications are scheduled together, we ran two time-sensitive processes with proportions of 40% and 20% and periods of 8192 μs and 512 μs respectively.[2] These processes were run first on an unloaded system to verify the correctness of the scheduler. Then, we evaluated the scheduler behavior when the same processes were run with competing file system load (described in Section 4.2.1). In this experiment each process runs a tight loop that repeatedly invokes the `gettimeofday` system call to measure the current time and stores this value in an array. The scheduler behavior is inferred at the user-level by simply measuring the time difference between successive elements of the array.

Table 1 shows the maximum (not the average) deviation in the proportion allocated and the period boundary for each of the two processes over the entire experiment. This table shows that the proportion-period scheduler allocates resources with a very low deviation of less than 25 μs on a lightly loaded system. Under heavy file system load the results show larger deviations. These deviations occur because execution time is "stolen" by the kernel interrupt handling code which runs at a higher priority than user-level processes in Linux. The maximum period deviation of 534 μs gives a lower bound on the latency tolerance of time-sensitive applications. For example, soft modems require periodic processing every 4 ms to 16 ms [8] and thus could be supported on TSL at the application level even under heavy file system load.

Figure 1 shows that while the maximum amount of time stolen is 490 μs and 20 μs for tasks 1 and 2, this time is over different periods. In particular, the interrupt handling code steals a maximum of 4-6% allocation time from the proportion-period processes. Note that this stolen time is due to interrupts other than the APIC timer interrupts since we maintain precise CPU cycle counter times for accounting purposes.

Currently, we provide the value of the stolen time to proportion-period applications or our feedback scheduler so that they can deal with it, for example by increasing thread allocations. This choice was motivated by

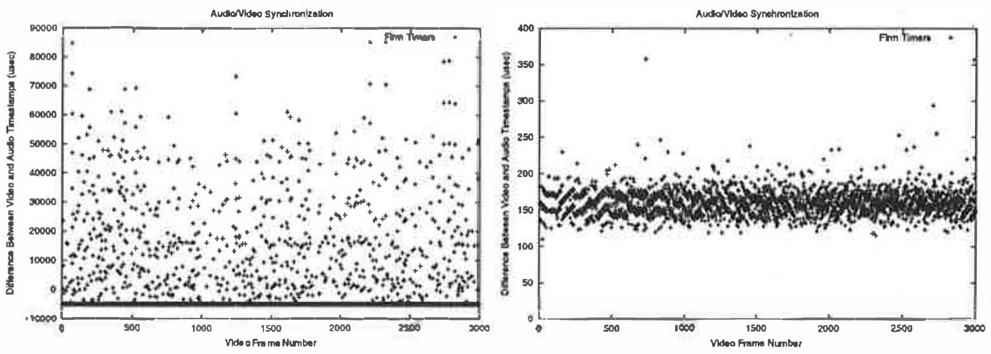
[2]The current proportion-period scheduler allows task periods that are multiples of 512 μs. While this period alignment restriction is not needed for a proportion-period scheduler, it simplifies feedback-based adjustment of task proportions.



a) Linux, X server non-real time (b) TSL, X server non-real time (c) TSL, X server real-time

Background load is a CPU stress test that run an empty loop. Note that the three figures have different scales, and that the maximum skew in Figure (c) is much smaller than the maximum skew in the other two cases.

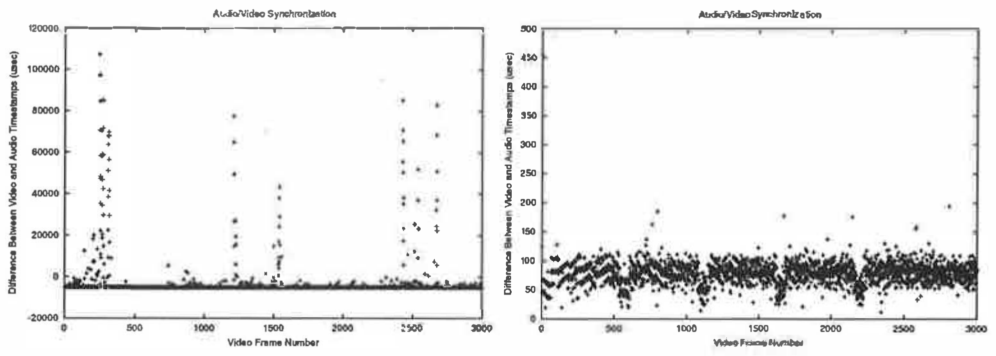
Figure 2: Audio/Video Skew on Linux and on TSL with user-level CPU load.



(a) Linux (b) TSL

Background load copies a 8 MB buffer from user level to a file with a single `write` call. Note that the two figures have different scales, and that the maximum skew in Figure (b) is much smaller than in Figure (a).

Figure 3: Audio/Video Skew on Linux and on TSL with kernel CPU load.



(a) Linux (b) TSL

Background load repeatedly copies a compiled Linux kernel sources directory recursively and then flushes the file system. Note that the two figures have different scales, and that the maximum skew in Figure (b) is much smaller than in Figure (a).

Figure 4: Audio/Video Skew on Linux and TSL with file-system load.

ric for mplayer. The latency metric for the proportion-period scheduler is maximum error in the allocation and period boundary.

4.2.1 Mplayer

Mplayer [14] is an audio/video player that can handle several different media formats. Mplayer synchronizes audio and video streams by using time-stamps that are associated with the audio and video frames. The audio card is used as a timing source and when a video frame is decoded, its time-stamp is compared with the time-stamp of the currently playing audio sample. If the video time-stamp is smaller than the audio time-stamp then the video frame is late and the video is immediately displayed. Otherwise, the system sleeps until the time difference between the video and audio time-stamps and then displays the video.

On a responsive kernel with sufficient available CPU capacity, audio/video synchronization can be achieved by simply sleeping for the correct amount of time. Unfortunately, if the kernel is unresponsive or has a coarse timing mechanism, mplayer will not be able to sleep for the correct amount of time leading to poor audio/video synchronization and high jitter in the inter-frame display times. Synchronization skew and display jitter are correlated and hence this paper only presents results for audio/video synchronization skew.

We compare the audio/video skew of mplayer between standard Linux and TSL under three competing loads: 1) non-kernel CPU load, 2) kernel CPU load, and 3) file-system load. For non-kernel load, a user-level CPU stress test is run in the background. For kernel CPU load, a large memory buffer is copied to a file, where the kernel uses CPU to move data from the user to the kernel space. Standard Linux does this activity in a non-preemptible section. This load spends 90% of its execution time in kernel mode. For the file system load, a large directory is copied recursively and the file system is flushed multiple times to create heavy file system activity. In each of these tests, mplayer is run for 100 seconds at real-time priority.

Non-kernel CPU load Figure 2 shows the audio/video skew in mplayer on Linux and on TSL when a CPU stress test is the competing load. This competing load runs an infinite loop consuming as much CPU as possible. Figure 2(a) shows that for standard Linux the maximum skew is large and ranges from -5 ms to 50 ms when the X server is run at a non-real time priority. Although not shown here, mplayer compensates for Linux's 10 ms timer resolution, and hence the skew under Linux without any competing load lies between -5 ms to 5 ms. Figure 2(b) shows that the skew for TSL, when X is run at a non-real time priority, is still as large as 35 ms. However, in this case, mplayer does not have to compensate for coarse timer resolution and thus most skew points lie close to 0 ms. Finally, Figure 2(c) shows that the skew for TSL improves considerably and is less than 250 us when the X server runs at real-time priority. The real-time priority value of X is the same as the priority assigned to mplayer.

These figures show that TSL works well on a non-kernel CPU load as long as the HLP protocol, described in Section 3.3.2, is used to assign priorities to time-sensitive tasks and to server tasks with the shared resources. Note that Linux with the X server at real-time priority still has a skew between -5 ms to 5 ms because of the timer resolution (not shown here).

As a result of this experiment, the rest of the experiments in this section are run with both mplayer and X at real-time priority to avoid any user-level priority inversion effects.

Kernel CPU Load The second experiment compares the audio/video skew in mplayer between Linux and TSL when the background load copies a large 8 MB memory buffer to a file with a single `write` system call. Figure 3(a) shows that the audio/video skew is as large as 90000 μs for Linux. In this case, the kernel moves the data from the user to the kernel space in a non-preemptible section. Figure 3(b) shows that the maximum skew is less than 400 μs for TSL. This improvement occurs as a result of improved kernel preemptibility for large write calls in TSL.

File System Load The third experiment compares the audio/video skew in mplayer between Linux and TSL when the background load repeatedly copies a compiled Linux kernel sources directory recursively and then flushes the file system. This directory has 13000 files and 180 MB of data and is stored on the Linux `ext2` file system. The kernel uses DMA for transferring disk data. Figure 4(a) shows that the skew under Linux can be as high as 12000 μs while Figure 4(b) shows that skew is less than 500 us under TSL. This result shows that TSL can provide low latencies even under heavy file-system and disk load.

highest priority among all time-sensitive clients accessing it. In this way, the X server cannot be preempted by the medium priority task.

The HLP protocol is very general and works with across multiple servers. Interestingly, this protocol handles the FIFO ordering problem in server queues mentioned in Section 3. Since servers have the highest priority among all their potential clients, they are able to serve each request immediately after it is enqueued and thus the queue size is never more than one and the queuing strategy is not relevant. After servicing the request, the next highest-priority client is scheduled and the latency caused by the server is minimized.

3.3.3 TSL Scheduling Model

If fixed priority tasks are the only ones accessing the X server, then the server can be scheduled with the maximum fixed priority, but in the background with respect to the proportion-period tasks. If, on the other hand, proportion-period tasks require access to the X server, then it must be scheduled with the highest priority in the system. This is the exception to the rule of scheduling fixed priority tasks in the background with respect to proportion-period tasks. Due of this exception, the shared server can jeopardize the proportion-period guarantee. Hence, the server must be "trusted", i.e., its execution time must be known or at least bounded.

Using real-time terminology, the shared server causes blocking time on proportion-period tasks. If this blocking time is known, it can be accounted in the guarantee of proportion-period tasks [19, 3]. Otherwise, the only safe thing to do is to leave some "unreserved CPU time". Hence, the admission test for proportion-period tasks is not $\sum_i P_i \leq 1$, but $\sum_i P_i \leq U^{max} < 1$. We have found that $U^{max} = 0.9$ is a reasonable value and works well in practice.[1]

4 Evaluation

This section describes the results of experiments we performed to evaluate 1) the behavior of time-sensitive applications running on TSL, and 2) the overheads of TSL. In these experiments TSL is derived from Linux version 2.4.16. It incorporates our firm timers, Robert Love's lock-breaking preemptible kernel patch and our proportion-period scheduler. Our experiments focus on evaluating the behavior of realistic time-sensitive applications running on a loaded general-purpose environment, and were run on a 1.5 GHz Pentium-4 Intel processor with 512 MB of memory.

[1]Note that if the blocking times are unknown it is impossible to provide a hard guarantee.

4.1 Micro Benchmarks

Before evaluating the impact of the latency reduction techniques used in TSL on real applications, we performed micro-benchmarks for evaluating the components of kernel latency, as described in Section 2. These components consist of timer latency, preemption latency and scheduling latency. We evaluated the first two components in isolation by running a time-sensitive process that needs to sleep for a specified amount of time (using the `nanosleep()` system call) and measures the time that it actually sleeps. In our first set of experiments, we evaluated timer latency and showed that it is 10 ms in standard Linux while firm timers reduce it to a few microseconds on TSL.

Next, we evaluated preemption latency when a number of different system loads are run in the background. We compared preemption latency under Linux, Andrew Morton's Linux with explicit preemption [13] and Robert Love's preemptible kernel and lock-breaking preemptible kernels [11]. The first interesting result was that on standard Linux the worst case preemption latency can be larger than 100 ms (when the kernel copies large amounts of data between kernel and user space) but in the common case preemption latency is less than 10 ms and is generally hidden by timer latency. However, when firm timers are used, preemption latency becomes more visible, and it is easy for latencies to be larger than 5 ms. Using the explicit preemption and kernel preemptibility techniques described in Section 3.2, preemption latency can be greatly reduced, and thus TSL provides a maximum kernel latency of less than 1 ms on our test machine even when the system is heavily loaded. The full details of these experiments and more results are presented in our previous paper [1], which we have briefly summarized here for the reader's convenience.

4.2 Latency in Real Applications

After evaluating kernel latency in isolation through micro-benchmarks, we performed experiments on two real applications, mplayer and our proportion-period scheduler which is a kernel-level application. We choose audio/video synchronization skew as the latency met-structures within spinlocks. This approach is used by Robert Love's lock-breaking preemptible kernel patch for Linux [11].

Our previous evaluation [1] shows that these approaches work fairly well for reducing preemption latency and should be incorporated in the design of any responsive kernel. As expected, the combined preemption approach is slightly superior to the first two approaches and, consequently, we have incorporated Robert Love's lock-breaking preemptible kernel patch in TSL for improving kernel responsiveness. Our experiments with real applications on TSL in Section 4.2 show that a responsive kernel complements an accurate timing mechanism to help improve time-sensitive application performance.

3.3 CPU Scheduling

The CPU scheduling algorithm should ensure that time-sensitive tasks obtain their correct allocation with low scheduling latency. We use a combination of the proportion-period and priority models, described in Section 2, to schedule time-sensitive applications. The proportion-period model provides temporal protection to applications and allows balancing the needs of time-sensitive applications with non-real time applications but requires specification of proportion and period scheduling parameters of each task. The priority model has a simpler programming interface and provides compatibility with applications based on a POSIX scheduler, but assumes that the timing needs of tasks are well-behaved.

3.3.1 Proportion-Period CPU Scheduling

For a single independent task, the simplest scheduling solution is to assign the highest priority to the task. However, with this solution, a misbehaving task that does not yield the CPU can starve all other tasks in the system. A commodity time-sensitive system should provide *temporal protection* to tasks so that misbehaved tasks that consume "too much" execution time do not affect the schedule of other tasks.

The proportion-period allocation model automatically provides temporal protection because each task is allocated a fixed proportion of the CPU at each task period. The period of a task is related to some application-level delay requirement of the application, such as the period or the jitter requirements of a task. The proportion is the amount of CPU allocation required every period for correct task execution. The proportion-period model can be effectively implemented using well known results from real-time scheduling research [10, 19].

We implemented a proportion-period CPU scheduler in Linux by using real-time scheduling techniques (EDF priority assignment) and by allowing a task to execute as a real-time task for a time Q every period T. The reserved time Q is equal to the product of the task's proportion P and its period T, and the end of each period is used as a deadline. After executing for a time Q, the task is blocked (or scheduled as a non real-time task) until its next period. When two tasks have the same deadline, the one with the smallest remaining capacity is scheduled to reduce the average finishing time.

The proportion-period model and its variants have been implemented and used extensively in the past [9]. While several of these schedulers are more sophisticated that ours, the main focus of this paper is to show that these schedulers can be implemented accurately in TSL and hence can improve the accuracy of scheduling analysis.

3.3.2 Priority CPU Scheduling

In the priority model, real-time priorities are assigned to time-sensitive tasks based on application needs [10]. Since the fixed priority model does not provide temporal protection, TSL schedules fixed priority tasks in the background with respect to proportion-period tasks. The only exception to this rule is with shared server tasks because they can cause *priority inversion*, and we must use a proper resource sharing protocol to bound its effects. In general, priority inversion occurs when an application is composed of multiple tasks that are interdependent. For example, consider a simple example of a video application consisting of a client and an X server. Let us assume that the client has been assigned the highest priority because it is time-sensitive. It displays graphics by requesting services from the X server. When it needs to display a video frame, it sends the frame to the server and then blocks waiting for the display to complete. If the X server has a priority lower than the client's priority, then it can be preempted by another task with a medium priority. Hence the medium priority task is delaying the server and thus delaying the high-priority client task.

We use a variant of the priority ceiling protocol [19] called the highest locking priority (HLP) protocol to cope with priority inversion. The HLP protocol works as follows: when a task acquires a resource, it automatically gets the highest priority of any task that can acquire this resource. In the example above, the display is the shared resource and thus the X server must have the

To derive the data structure efficiency benefits of periodic timers, firm timers combine the periodic timing mechanism with the one-shot timing mechanism for timers that need a timeout longer than the period of the periodic timer interrupt. A firm timer for a long timeout uses a periodic timer to wake up at the last period before the timer expiration and then sets the one-shot APIC timer. Consequently, our firm timers approach only has active one-shot timers within one tick period. Since the number of such timers, *n*, is decreased, the data structure implementation becomes more efficient. Note that operating systems generally perform periodic activity such as time keeping, accounting and profiling at each periodic tick interrupt and thus the dual wakeup does not add any additional cost. An additional benefit of this approach is that timer drift in firm timers is limited to a small fraction of the period of the periodic timer interrupt assuming, as in Linux, that the periodic interrupt is synchronized to the global NTP protocol while the APIC timer is not.

The firm timer expiration times are specified as CPU clock cycle values. In an x86 processor, the current time in CPU cycles in stored in a 64 bit register. Timer expiration values can be stored as 64 bit quantities also but this choice involves expensive 64 bit time conversions from CPU cycles to memory cycles needed for programming the APIC timer. A more efficient alternative for time conversion is to store the expiration times as 32 bit quantities. However, this approach leads to quick roll over on modern CPUs. For example, on a two GHz processor, 32 bits roll over every second. Fortunately, firm timers are still able to use 32 bit expiration times because they use periodic timers for long timeouts and use one-shot timer expiration values only within a periodic tick.

We want to provide the benefits of the firm timer accurate timing mechanism to standard user-level applications. These applications use the standard POSIX interface calls such as `nanosleep()`, `pause()`, `setitimer()`, `select()` and `poll()`. We have modified the implementation of these system calls in TSL to use firm timers without changing the interface of these calls. As a result, unmodified applications automatically get increased timer accuracy in our system as shown in Section 4.2.

3.2 Fine-Grained Kernel Preemptibility

A kernel is responsive when its non-preemptible sections, which keep the scheduler from being invoked to schedule a task, are small. There are two main reasons why the scheduler may not be able to run. One is that interrupts might be disabled. For example, if the timer interrupt in Figure 1 is disabled, the timer task can only enter the ready queue when the interrupt is re-enabled. Another, potentially more significant reason is that another thread may be executing in a critical section in the kernel. For example, the timer task upon entering the ready queue will be scheduled only when the other thread exits its non-preemptible critical section.

The length of non-preemptible sections in a kernel depends on the strategy that the kernel uses to guarantee the consistency of its internal structures and on the internal organization of the kernel. Traditional commodity kernels disable preemption for the entire period of time when a thread is in the kernel, i.e., when an interrupt fires or for the duration of a system call, except before certain well-known long operations are invoked by the kernel. For example, they generally allow preemption before invoking disk I/O operations. Unfortunately, with this structure, preemption latency under standard Linux can be greater than 30 ms [1].

One approach that reduces preemption latency is explicit insertion of preemption points at strategic points inside the kernel so that a thread in the kernel explicitly yields the CPU to the scheduler when it reaches these preemption points. In this way, the size of non-preemptible sections is reduced. The choice of preemption points depends on system call paths and has to be manually placed after careful auditing of system code. This approach is used by some real-time versions of Linux, such as RED Linux [17] and by Andrew Morton's low-latency project [13]. Preemption latency in such a kernel decreases to the maximum time between two preemption points.

Another approach, used in most real-time systems, is to allow preemption anytime the kernel is not accessing shared data structures. To support this fine level of kernel preemptibility, shared kernel data must be explicitly protected using mutexes or spinlocks. The Linux preemptible kernel project [11] uses this approach and disables kernel preemption only when a spinlock is held or an interrupt handler is executing to protect the consistency of shared kernel data structures. In a preemptible kernel, preemption latency is determined by the maximum amount of time for which a spinlock is held inside the kernel and the maximum time taken by interrupt service routines.

The preemptible kernel approach can have high preemption latency when spinlocks are held for a long time. To address this problem, a third approach combines explicit preemption with the preemptible kernel approach by releasing (and reacquiring) spin-locks at strategic points in long code sections that access shared data ample, reprogramming the standard programmable interval timer (PIT) on Intel x86 is expensive because it requires several slow `out` instructions on the ISA bus. In contrast, our firm-timers implementation uses the APIC one-shot timer present in newer Intel Pentium class machines. This timer resides on-chip and can be reprogrammed in a few cycles without any noticeable performance penalty.

Since timer reprogramming is inexpensive, the key overhead for the one-shot timing mechanism in firm timers lies in fielding interrupts. Interrupts are asynchronous events that cause an uncontrolled context switch and result in cache pollution. To avoid interrupts, firm timers use soft timers, which poll for expired timers at strategic points in the kernel such as at system call, interrupt, and exception return paths. At these points, the working set in the cache is likely to be replaced anyway and hence polling and dispatching timers does not cause significant additional overhead. In essence, soft timers allow voluntary switching of context at "convenient" moments.

While soft timers reduce the costs associated with interrupt handling, they introduce two new problems. First, there is a cost in polling or checking for timers at each soft-timer point. Later, in Section 4.3.2, we analyze this cost in detail and show that it can be amortized if a certain percentage of checks result in the firing of timers. Second, this polling approach introduces timer latency when the checks occur infrequently or the distribution of the checks and the timer deadlines are not well matched.

Firm timers avoid the second problem by combining one-shot timers with soft timers by exposing a system-wide *timer overshoot* parameter. With this parameter, the one-shot timer is programmed to fire an overshoot amount of time after the next timer expiry (instead of exactly at the next timer expiry). In some cases, an interrupt, system call, or exception may happen after a timer has expired but before the one-shot APIC timer generates an interrupt. At this point, the timer expiration is handled and the one-shot APIC timer is again reprogrammed an overshoot amount of time after the next timer expiry event. When soft-timers are effective, firm timers repeatedly reprogram the one-shot timer for the next timer expiry but do not incur the overhead associated with fielding interrupts.

The timer overshoot parameter allows making a tradeoff between accuracy and overhead. A small value of timer overshoot provides high timer resolution but increases overhead since the soft timing component of firm timers are less likely to be effective. Conversely, a large value decreases timer overhead at the cost of increased maximum timer latency. The overshoot value can be changed dynamically. With a zero value, we obtain one-shot timers (or hard timers) and with a large value, we obtain soft timers. A choice in between leads to our hybrid firm timers approach. This choice depends on the timing accuracy needed by applications.

3.1.2 Firm Timers Implementation

Firm timers in TSL maintain a timer queue for each processor. The timer queue is kept sorted by timer expiry. The one-shot APIC timer is programmed to generate an interrupt at the next timer expiry event. When the APIC timer expires, the interrupt handler checks the timer queue and executes the callback function associated with each expired timer in the queue. Expired timers are removed while periodic timers are re-enqueued after their expiration field is incremented by the value in their period field. The APIC timer is then reprogrammed to generate an interrupt at the next timer event.

The APIC is set by writing a value into a register which is decremented at each memory bus cycle until it reaches zero and generates an interrupt. Given a 100 MHz memory bus available on a modern machine, a one-shot timer has a theoretical accuracy of 10 nanoseconds. However, in practice, the time needed to field timer interrupts is significantly higher and is the limiting factor for timer accuracy.

Soft timers are enabled by using a non-zero timer overshoot value, in which case, the APIC timer is set an overshoot amount after the next timer event. Our current implementation uses a single global overshoot value. It is possible to extend this implementation so that each timer or an application using this timer can specify its desired overshoot or timing accuracy. In this case, only applications with tighter timing constraints cause the additional interrupt cost of more precise timers. The overhead in this alternate implementation involves keeping an additional timer queue sorted by the timer expiry plus overshoot value.

The data structures for one-shot timers are less efficient than for periodic timers. For instance, periodic timers can be implemented using calendar queues [5] which operate in $O(1)$ time, while one-shot timers require priority heaps which require $O(log(n))$ time, where n is the number of active timers. This difference exists because periodic timers have a natural bucket width (in time) that is the period of the timer interrupt. Calendar queues need this fixed bucket width and derive their efficiency by providing no ordering to timers within a bucket. One-shot fine-grained timers have no corresponding bucket width.

these solutions have generally not been integrated: on one hand, real-time research has developed good schedulers and analyzed them from a mathematical point of view, and on the other hand, there are real systems that provide a responsive kernel but provide simplistic schedulers that are only designed to be fast and efficient [12]. Real-time operating systems integrate these solutions for time-sensitive tasks but tend to ignore the performance overhead of their solutions on throughput-oriented applications [17]. Our goal is to support both types of applications well.

It is worth noting that latency due to system services, such as the X server for graphical display [21] on a Linux system, has the same components of kernel latency described above. In fact, the simple scheduling models presented above assume that tasks are independent. In a real application, tasks can be interdependent which can cause priority inversion problems [15]. For example, in Section 4.2.1 we show that a multimedia player that uses the X server for display can perform sub-optimally due to priority inversion, even if the kernel allocates resources correctly. The X server operates on client requests in an event-driven manner and the handling of each event is non-preemptible and generally in FIFO order. As a result, time-sensitive clients expecting service from the server observe latencies that depend on the time to service previous client requests: the performance of time-sensitive applications depends on not just kernel support for such applications but also the design of other system services. Thus in Section 3.3, we enhance the priority scheduler with techniques that solve priority inversion.

In the next section, we present Time-Sensitive Linux (TSL) that provides accurate timers, a responsive kernel, and time-sensitive scheduling algorithms to support the requirements highlighted above.

3 Implementing Time-Sensitive Linux

We propose three specific techniques, firm timers, fine-grained kernel preemptibility and proportion-period CPU scheduling for reducing the three components of kernel latency. We have integrated these techniques in the Linux kernel to implement TSL.

3.1 Firm Timers

Firm timers provide an accurate timing mechanism with low overhead by exploiting the benefits associated with three different approaches for implementing timers: one-shot timers, soft timers and periodic timers.

Traditionally, commodity operating systems have implemented their timing mechanism with periodic timers. These timers are normally implemented with periodic timer interrupts. For example, on Intel x86 machines, these interrupts are generated by the Programmable Interval Timer (PIT), and on Linux, the period of these interrupts is 10 ms. As a result, the maximum timer latency is 10 ms. This latency can be reduced by reducing the period of the timer interrupt but this solution increases system overhead because the timer interrupts are generated more frequently.

To reduce the overhead of timers, it is necessary to move from a periodic timer interrupt model to a one-shot timer interrupt model where interrupts are generated only when needed. Consider two tasks with periods 5 and 7 ms. With periodic timers and a period of 1 ms, the maximum timer latency would be 1 ms. In addition, in 35 ms, 35 interrupts would be generated. With one-shot timers, interrupts will be generated at 5 ms, 7 ms, 10 ms, etc., and the total number of interrupts in 35 ms is 11. Also, the timer latency will be close to the interrupt service time, which is relatively small. Hence, one-shot timers avoid unnecessary interrupts and reduce timer latency.

3.1.1 Firm Timers Design

Firm timers, at their core, use one-shot timers for efficient and accurate timing. One-shot timers generate a timer interrupt at the next timer expiry. At this time, expired timers are dispatched and then finally the timer interrupt is reprogrammed for the next timer expiry. Hence, there are two main costs associated with one-shot timers, timer reprogramming and fielding timer interrupts. Unlike periodic timers, one-shot timers have to be continuously reprogrammed for each timer event. More importantly, as the frequency of timer events increases, the interrupt handling overhead grows until it limits timer frequency. To overcome these challenges, firm timers use inexpensive reprogramming available on modern hardware and combine soft timers (originally proposed by Aron and Druschel [2]) with one-shot timers to reduce the number of hardware generated timer interrupts. Below, we discuss these points in more detail.

While timer reprogramming on traditional hardware has been expensive (and has thus encouraged using periodic timers), it has become inexpensive on modern hardware such as Intel Pentium II and later machines. For ex-

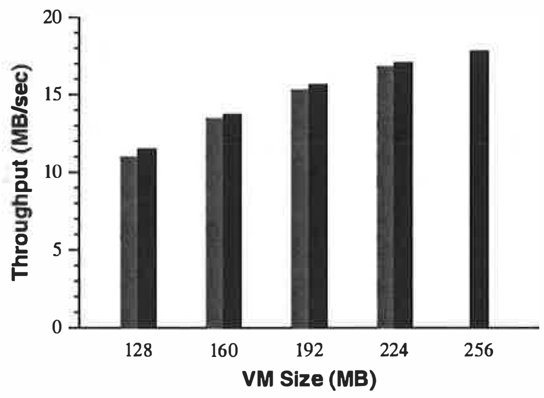


Figure 2: **Balloon Performance.** Throughput of single Linux VM running dbench with 40 clients. The black bars plot the performance when the VM is configured with main memory sizes ranging from 128 MB to 256 MB. The gray bars plot the performance of the same VM configured with 256 MB, ballooned down to the specified size.

or removed from a VM in order to rapidly adjust its physical memory size.

To demonstrate the effectiveness of ballooning, we used the synthetic dbench benchmark [28] to simulate fileserver performance under load from 40 clients. This workload benefits significantly from additional memory, since a larger buffer cache can absorb more disk traffic. For this experiment, ESX Server was running on a dual-processor Dell Precision 420, configured to execute one VM running Red Hat Linux 7.2 on a single 800 MHz Pentium III CPU.

Figure 2 presents dbench throughput as a function of VM size, using the average of three consecutive runs for each data point. The ballooned VM tracks non-ballooned performance closely, with an observed overhead ranging from 4.4% at 128 MB (128 MB balloon) down to 1.4% at 224 MB (32 MB balloon). This overhead is primarily due to guest OS data structures that are sized based on the amount of "physical" memory; the Linux kernel uses more space in a 256 MB system than in a 128 MB system. Thus, a 256 MB VM ballooned down to 128 MB has slightly less free space than a VM configured with exactly 128 MB.

Despite its advantages, ballooning does have limitations. The balloon driver may be uninstalled, disabled explicitly, unavailable while a guest OS is booting, or temporarily unable to reclaim memory quickly enough to satisfy current system demands. Also, upper bounds on reasonable balloon sizes may be imposed by various guest OS limitations.

3.3 Demand Paging

ESX Server preferentially uses ballooning to reclaim memory, treating it as a common-case optimization. When ballooning is not possible or insufficient, the system falls back to a paging mechanism. Memory is reclaimed by paging out to an ESX Server swap area on disk, without any guest involvement.

The ESX Server swap daemon receives information about target swap levels for each VM from a higher-level policy module. It manages the selection of candidate pages and coordinates asynchronous page outs to a swap area on disk. Conventional optimizations are used to maintain free slots and cluster disk writes.

A randomized page replacement policy is used to prevent the types of pathological interference with native guest OS memory management algorithms described in Section 3.1. This choice was also guided by the expectation that paging will be a fairly uncommon operation. Nevertheless, we are investigating more sophisticated page replacement algorithms, as well policies that may be customized on a per-VM basis.

4 Sharing Memory

Server consolidation presents numerous opportunities for sharing memory between virtual machines. For example, several VMs may be running instances of the same guest OS, have the same applications or components loaded, or contain common data. ESX Server exploits these sharing opportunities, so that server workloads running in VMs on a single machine often consume less memory than they would running on separate physical machines. As a result, higher levels of overcommitment can be supported efficiently.

4.1 Transparent Page Sharing

Disco [3] introduced *transparent page sharing* as a method for eliminating redundant copies of pages, such as code or read-only data, across virtual machines. Once copies are identified, multiple guest "physical" pages are mapped to the same machine page, and marked copy-on-write. Writing to a shared page causes a fault that generates a private copy.

Unfortunately, Disco required several guest OS modifications to identify redundant copies as they were created. For example, the bcopy() routine was hooked to enable file buffer cache sharing across virtual machines. Some sharing also required the use of non-standard or restricted interfaces. A special network interface with support for large packets facilitated sharing data communicated between VMs on a virtual subnet. Interposition on disk accesses allowed data from shared, non-persistent disks to be shared across multiple guests.

4.2 Content-Based Page Sharing

Because modifications to guest operating system internals are not possible in our environment, and changes to application programming interfaces are not acceptable, ESX Server takes a completely different approach to page sharing. The basic idea is to identify page copies by their contents. Pages with identical contents can be shared regardless of when, where, or how those contents were generated. This general-purpose approach has two key advantages. First, it eliminates the need to modify, hook, or even understand guest OS code. Second, it can identify more opportunities for sharing; by definition, all potentially shareable pages can be identified by their contents.

The cost for this unobtrusive generality is that work must be performed to scan for sharing opportunities. Clearly, comparing the contents of each page with every other page in the system would be prohibitively expensive; naive matching would require O(n^2) page comparisons. Instead, hashing is used to identify pages with potentially-identical contents efficiently.

A hash value that summarizes a page's contents is used as a lookup key into a hash table containing entries for other pages that have already been marked copy-on-write (COW). If the hash value for the new page matches an existing entry, it is very likely that the pages are identical, although false matches are possible. A successful match is followed by a full comparison of the page contents to verify that the pages are identical.

Once a match has been found with an existing shared page, a standard copy-on-write technique can be used to share the pages, and the redundant copy can be reclaimed. Any subsequent attempt to write to the shared page will generate a fault, transparently creating a private copy of the page for the writer.

If no match is found, one option is to mark the page COW in anticipation of some future match. However, this simplistic approach has the undesirable side-effect of marking *every* scanned page copy-on-write, incurring unnecessary overhead on subsequent writes. As an optimization, an unshared page is not marked COW, but instead tagged as a special *hint* entry. On any future match with another page, the contents of the hint page are rehashed. If the hash has changed, then the hint page has been modified, and the stale hint is removed. If the hash is still valid, a full comparison is performed, and the pages are shared if it succeeds.

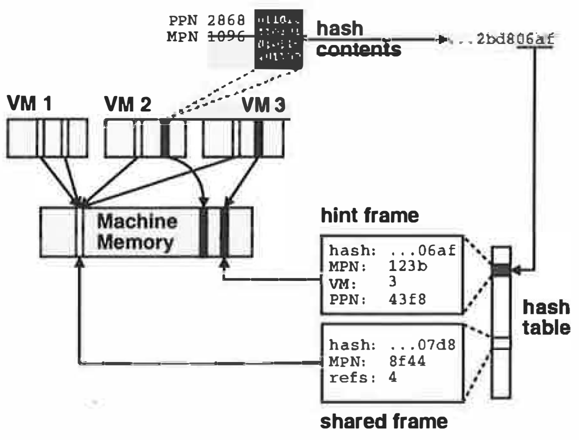


Figure 3: **Content-Based Page Sharing.** ESX Server scans for sharing opportunities, hashing the contents of candidate PPN 0x2868 in VM 2. The hash is used to index into a table containing other scanned pages, where a match is found with a hint frame associated with PPN 0x43f8 in VM 3. If a full comparison confirms the pages are identical, the PPN-to-MPN mapping for PPN 0x2868 in VM 2 is changed from MPN 0x1096 to MPN 0x123b, both PPNs are marked COW, and the redundant MPN is reclaimed.

Higher-level page sharing policies control when and where to scan for copies. One simple option is to scan pages incrementally at some fixed rate. Pages could be considered sequentially, randomly, or using heuristics to focus on the most promising candidates, such as pages marked read-only by the guest OS, or pages from which code has been executed. Various policies can be used to limit CPU overhead, such as scanning only during otherwise-wasted idle cycles.

4.3 Implementation

The ESX Server implementation of content-based page sharing is illustrated in Figure 3. A single global hash table contains frames for all scanned pages, and chaining is used to handle collisions. Each frame is encoded compactly in 16 bytes. A *shared frame* consists of a hash value, the machine page number (MPN) for the shared page, a reference count, and a link for chaining. A *hint frame* is similar, but encodes a truncated hash value to make room for a reference back to the corresponding guest page, consisting of a VM identifier and a physical page number (PPN). The total space overhead for page sharing is less than 0.5% of system memory.

Unlike the Disco page sharing implementation, which maintained a backmap for each shared page, ESX Server uses a simple reference count. A small 16-bit count is stored in each frame, and a separate overflow table is used to store any extended frames with larger counts. This allows highly-shared pages to be represented compactly. For example, the empty *zero page* filled completely with zero bytes is typically shared with a large reference count. A similar overflow technique for large reference counts was used to save space in the early OOZE virtual memory system [15].

A fast, high-quality hash function [14] is used to generate a 64-bit hash value for each scanned page. Since the chance of encountering a false match due to hash aliasing is incredibly small[3] the system can make the simplifying assumption that all shared pages have unique hash values. Any page that happens to yield a false match is considered ineligible for sharing.

The current ESX Server page sharing implementation scans guest pages randomly. Although more sophisticated approaches are possible, this policy is simple and effective. Configuration options control maximum per-VM and system-wide page scanning rates. Typically, these values are set to ensure that page sharing incurs negligible CPU overhead. As an additional optimization, the system always attempts to share a page before paging it out to disk.

To evaluate the ESX Server page sharing implementation, we conducted experiments to quantify its effectiveness at reclaiming memory and its overhead on system performance. We first analyze a "best case" workload consisting of many homogeneous VMs, in order to demonstrate that ESX Server is able to reclaim a large fraction of memory when the potential for sharing exists. We then present additional data collected from production deployments serving real users.

We performed a series of controlled experiments using identically-configured virtual machines, each running Red Hat Linux 7.2 with 40 MB of "physical" memory. Each experiment consisted of between one and ten concurrent VMs running SPEC95 benchmarks for thirty minutes. For these experiments, ESX Server was running on a Dell PowerEdge 1400SC multiprocessor with two 933 MHz Pentium III CPUs.

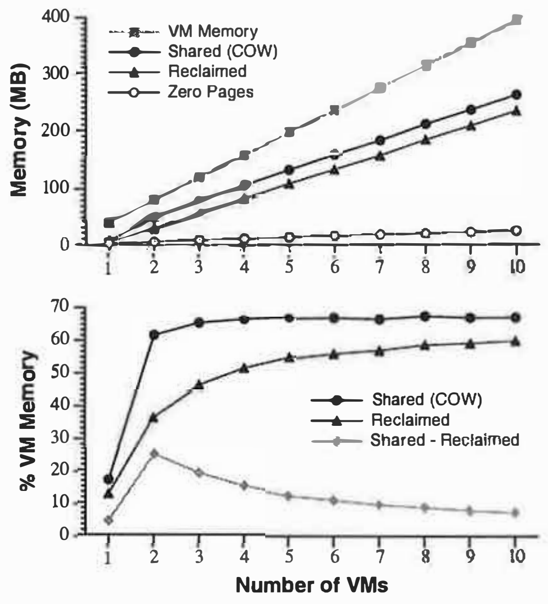


Figure 4: **Page Sharing Performance.** Sharing metrics for a series of experiments consisting of identical Linux VMs running SPEC95 benchmarks. The top graph indicates the absolute amounts of memory shared and saved increase smoothly with the number of concurrent VMs. The bottom graph plots these metrics as a percentage of aggregate VM memory. For large numbers of VMs, sharing approaches 67% and nearly 60% of all VM memory is reclaimed.

Figure 4 presents several sharing metrics plotted as a function of the number of concurrent VMs. Surprisingly, some sharing is achieved with only a single VM. Nearly 5 MB of memory was reclaimed from a single VM, of which about 55% was due to shared copies of the zero page. The top graph shows that after an initial jump in sharing between the first and second VMs, the total amount of memory shared increases linearly with the number of VMs, as expected. Little sharing is attributed to zero pages, indicating that most sharing is due to redundant code and read-only data pages. The bottom graph plots these metrics as a percentage of aggregate VM memory. As the number of VMs increases, the sharing level approaches 67%, revealing an overlap of approximately two-thirds of all memory between the VMs. The amount of memory required to contain the single copy of each common shared page (labelled Shared – Reclaimed), remains nearly constant, decreasing as a percentage of overall VM memory.

[3]Assuming page contents are randomly mapped to 64-bit hash values, the probability of a single collision doesn't exceed 50% until approximately $\sqrt{2^{64}} = 2^{32}$ distinct pages are hashed [14]. For a static snapshot of the largest possible IA-32 memory configuration with 2^{24} pages (64 GB), the collision probability is less than 0.01%.

		Total	Shared		Reclaimed	
	Guest Types	MB	MB	%	MB	%
A	10 WinNT	2048	880	42.9	673	32.9
B	9 Linux	1846	539	29.2	345	18.7
C	5 Linux	1658	165	10.0	120	7.2

Figure 5: **Real-World Page Sharing.** Sharing metrics from production deployments of ESX Server. (a) Ten Windows NT VMs serving users at a Fortune 50 company, running a variety of database (Oracle, SQL Server), web (IIS, Websphere), development (Java, VB), and other applications. (b) Nine Linux VMs serving a large user community for a nonprofit organization, executing a mix of web (Apache), mail (Majordomo, Postfix, POP/IMAP, MailArmor), and other servers. (c) Five Linux VMs providing web proxy (Squid), mail (Postfix, RAV), and remote access (ssh) services to VMware employees.

The CPU overhead due to page sharing was negligible. We ran an identical set of experiments with page sharing disabled, and measured no significant difference in the aggregate throughput reported by the CPU-bound benchmarks running in the VMs. Over all runs, the aggregate throughput was actually 0.5% *higher* with page sharing enabled, and ranged from 1.6% lower to 1.8% higher. Although the effect is generally small, page sharing does improve memory locality, and may therefore increase hit rates in physically-indexed caches.

These experiments demonstrate that ESX Server is able to exploit sharing opportunities effectively. Of course, more diverse workloads will typically exhibit lower degrees of sharing. Nevertheless, many real-world server consolidation workloads do consist of numerous VMs running the same guest OS with similar applications. Since the amount of memory reclaimed by page sharing is very workload-dependent, we collected memory sharing statistics from several ESX Server systems in production use.

Figure 5 presents page sharing metrics collected from three different production deployments of ESX Server. Workload A, from a corporate IT department at a Fortune 50 company, consists of ten Windows NT 4.0 VMs running a wide variety of database, web, and other servers. Page sharing reclaimed nearly a third of all VM memory, saving 673 MB. Workload B, from a nonprofit organization's Internet server, consists of nine Linux VMs ranging in size from 64 MB to 768 MB, running a mix of mail, web, and other servers. In this case, page sharing was able to reclaim 18.7% of VM memory, saving 345 MB, of which 70 MB was attributed to zero pages. Finally, workload C is from VMware's own IT department, and provides web proxy, mail, and remote access services to our employees using five Linux VMs ranging in size from 32 MB to 512 MB. Page sharing reclaimed about 7% of VM memory, for a savings of 120 MB, of which 25 MB was due to zero pages.

5 Shares vs. Working Sets

Traditional operating systems adjust memory allocations to improve some aggregate, system-wide performance metric. While this is usually a desirable goal, it often conflicts with the need to provide quality-of-service guarantees to clients of varying importance. Such guarantees are critical for server consolidation, where each VM may be entitled to different amounts of resources based on factors such as importance, ownership, administrative domains, or even the amount of money paid to a service provider for executing the VM. In such cases, it can be preferable to penalize a less important VM, even when that VM would derive the largest performance benefit from additional memory.

ESX Server employs a new allocation algorithm that is able to achieve efficient memory utilization while maintaining memory performance isolation guarantees. In addition, an explicit parameter is introduced that allows system administrators to control the relative importance of these conflicting goals.

5.1 Share-Based Allocation

In proportional-share frameworks, resource rights are encapsulated by *shares*, which are owned by *clients* that consume resources.[4] A client is entitled to consume resources proportional to its share allocation; it is guaranteed a minimum resource fraction equal to its fraction of the total shares in the system. Shares represent relative resource rights that depend on the total number of shares contending for a resource. Client allocations degrade gracefully in overload situations, and clients proportionally benefit from extra resources when some allocations are underutilized.

Both randomized and deterministic algorithms have been proposed for proportional-share allocation of space-shared resources. The dynamic *min-funding revocation* algorithm [31, 32] is simple and effective. When one client demands more space, a replacement algorithm selects a victim client that relinquishes some of its previously-allocated space. Memory is revoked from the

[4] Shares are alternatively referred to as *tickets* or *weights* in the literature. The term *clients* is used to abstractly refer to entities such as threads, processes, VMs, users, or groups.

client that owns the fewest shares per allocated page. Using an economic analogy, the *shares-per-page* ratio can be interpreted as a price; revocation reallocates memory away from clients paying a lower price to those willing to pay a higher price.

5.2 Reclaiming Idle Memory

A significant limitation of pure proportional-share algorithms is that they do not incorporate any information about active memory usage or working sets. Memory is effectively partitioned to maintain specified ratios. However, idle clients with many shares can hoard memory unproductively, while active clients with few shares suffer under severe memory pressure. In general, the goals of performance isolation and efficient memory utilization often conflict. Previous attempts to cross-apply techniques from proportional-share CPU resource management to compensate for idleness have not been successful [25].

ESX Server resolves this problem by introducing an *idle memory tax.* The basic idea is to charge a client more for an idle page than for one it is actively using. When memory is scarce, pages will be reclaimed preferentially from clients that are not actively using their full allocations. The tax rate specifies the maximum fraction of idle pages that may be reclaimed from a client. If the client later starts using a larger portion of its allocated memory, its allocation will increase, up to its full share.

Min-funding revocation is extended to use an adjusted shares-per-page ratio. For a client with S shares and an allocation of P pages, of which a fraction f are active, the adjusted shares-per-page ratio ρ is

$$\rho = \frac{S}{P \cdot (f + k \cdot (1 - f))}$$

where the idle page cost $k = 1/(1 - \tau)$ for a given tax rate $0 \leq \tau < 1$.

The tax rate τ provides explicit control over the desired policy for reclaiming idle memory. At one extreme, $\tau = 0$ specifies pure share-based isolation. At the other, $\tau \approx 1$ specifies a policy that allows all of a client's idle memory to be reclaimed for more productive uses.

The ESX Server idle memory tax rate is a configurable parameter that defaults to 75%. This allows most idle memory in the system to be reclaimed, while still providing a buffer against rapid working set increases, masking the latency of system reclamation activity such as ballooning and swapping.[5]

5.3 Measuring Idle Memory

For the idle memory tax to be effective, the server needs an efficient mechanism to estimate the fraction of memory in active use by each virtual machine. However, specific active and idle pages need not be identified individually.

One option is to extract information using native interfaces within each guest OS. However, this is impractical, since diverse activity metrics are used across various guests, and those metrics tend to focus on per-process working sets. Also, guest OS monitoring typically relies on access bits associated with page table entries, which are bypassed by DMA for device I/O.

ESX Server uses a statistical sampling approach to obtain aggregate VM working set estimates directly, without any guest involvement. Each VM is sampled independently, using a configurable sampling period defined in units of VM execution time. At the start of each sampling period, a small number n of the virtual machine's "physical" pages are selected randomly using a uniform distribution. Each sampled page is tracked by invalidating any cached mappings associated with its PPN, such as hardware TLB entries and virtualized MMU state. The next guest access to a sampled page will be intercepted to re-establish these mappings, at which time a touched page count t is incremented. At the end of the sampling period, a statistical estimate of the fraction f of memory actively accessed by the VM is $f = t/n$.

The sampling rate may be controlled to tradeoff overhead and accuracy. By default, ESX Server samples 100 pages for each 30 second period. This results in at most 100 minor page faults per period, incurring negligible overhead while still producing reasonably accurate working set estimates.

Estimates are smoothed across multiple sampling periods. Inspired by work on balancing stability and agility from the networking domain [16], we maintain separate exponentially-weighted moving averages with different gain parameters. A slow moving average is used to produce a smooth, stable estimate. A fast moving average

[5]The configured tax rate applies uniformly to all VMs. While the underlying implementation supports separate, per-VM tax rates, this capability is not currently exposed to users. Customized or graduated tax rates may be useful for more sophisticated control over relative allocations and responsiveness.

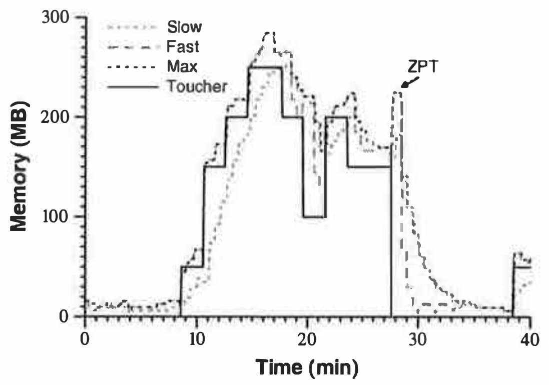


Figure 6: **Active Memory Sampling.** A Windows VM executes a simple memory *toucher* application. The solid black line indicates the amount of memory repeatedly touched, which is varied over time. The dotted black line is the sampling-based statistical estimate of overall VM memory usage, including background Windows activities. The estimate is computed as the *max* of *fast* (gray dashed line) and *slow* (gray dotted line) moving averages. The spike labelled *ZPT* is due to the Windows "zero page thread."

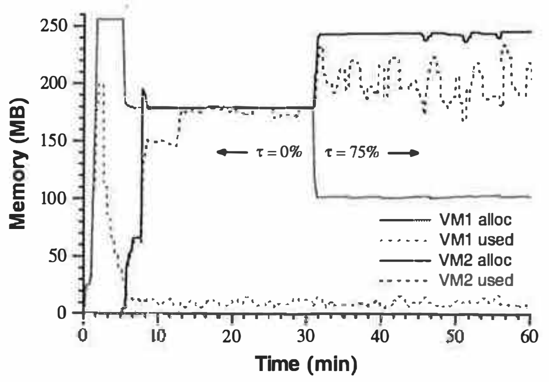


Figure 7: **Idle Memory Tax.** Two VMs with identical share allocations are each configured with 256 MB in an overcommitted system. VM1 (gray) runs Windows, and remains idle after booting. VM2 (black) executes a memory-intensive Linux workload. For each VM, ESX Server allocations are plotted as solid lines, and estimated memory usage is indicated by dotted lines. With an initial tax rate of 0%, the VMs each converge on the same 179 MB allocation. When the tax rate is increased to 75%, idle memory is reclaimed from VM1 and reallocated to VM2, boosting its performance by over 30%.

adapts quickly to working set changes. Finally, a version of the fast average that incorporates counts from the current sampling period is updated incrementally to reflect rapid intra-period changes.

The server uses the maximum of these three values to estimate the amount of memory being actively used by the guest. This causes the system to respond rapidly to increases in memory usage and more gradually to decreases in memory usage, which is the desired behavior. A VM that had been idle and starts using memory is allowed to ramp up to its share-based allocation quickly, while a VM that had been active and decreases its working set has its idle memory reclaimed slowly via the idle memory tax.

5.4 Experimental Results

This section presents quantitative experiments that demonstrate the effectiveness of memory sampling and idle memory taxation. Memory sampling is used to estimate the fraction of memory actively used by each VM. These estimates are then incorporated into the idle memory tax computations performed by the share-based memory allocation algorithm.

Figure 6 presents the results of an experiment designed to illustrate the memory sampling technique. For this experiment, ESX Server was running on a dual-processor Dell Precision 420, configured to execute one VM running Windows 2000 Advanced Server on a single 800 MHz Pentium III CPU.

A user-level *toucher* application allocates and repeatedly accesses a controlled amount of memory that is varied between 50 MB and 250 MB. An additional 10–20 MB is accessed by standard Windows background activities. As expected, the statistical estimate of active memory usage responds quickly as more memory is touched, tracking the fast moving average, and more slowly as less memory is touched, tracking the slow moving average.

We were originally surprised by the unexpected spike immediately after the toucher application terminates, an effect that does not occur when the same experiment is run under Linux. This is caused by the Windows "zero page thread" that runs only when no other threads are runnable, clearing the contents of pages it moves from the free page list to the zeroed page list [22].

Figure 7 presents experimental results that demonstrate the effectiveness of imposing a tax on idle memory. For this experiment, ESX Server was running on a Dell Precision 420 multiprocessor with two 800 MHz Pentium III CPUs and 512 MB RAM, of which approximately 360 MB was available for executing VMs.[6]

[6]Some memory is required for per-VM virtualization overheads, which are discussed in Section 6.2. Additional memory is required for ESX Server itself; the smallest recommended configuration is 512 MB.

Two VMs with identical share allocations are each configured with 256 MB "physical" memory. The first VM that powers on runs Windows 2000 Advanced Server, and remains idle after booting. A few minutes later, a second VM is started, running a memory-intensive `dbench` workload [28] under Red Hat Linux 7.2. The initial tax rate is set to $\tau = 0$, resulting in a pure share-based allocation. Despite the large difference in actual memory usage, each VM receives the same 179 MB allocation from ESX Server. In the middle of the experiment, the tax rate is increased to $\tau = 0.75$, causing memory to be reclaimed from the idle Windows VM and reallocated to the active Linux VM running `dbench`. The `dbench` workload benefits significantly from the additional memory, increasing throughput by over 30% after the tax rate change.

6 Allocation Policies

ESX Server computes a target memory allocation for each VM based on both its share-based entitlement and an estimate of its working set, using the algorithm presented in Section 5. These targets are achieved via the ballooning and paging mechanisms presented in Section 3. Page sharing runs as an additional background activity that reduces overall memory pressure on the system. This section describes how these various mechanisms are coordinated in response to specified allocation parameters and system load.

6.1 Parameters

System administrators use three basic parameters to control the allocation of memory to each VM: a *min* size, a *max* size, and memory *shares*. The *min* size is a guaranteed lower bound on the amount of memory that will be allocated to the VM, even when memory is overcommitted. The *max* size is the amount of "physical" memory configured for use by the guest OS running in the VM. Unless memory is overcommitted, VMs will be allocated their *max* size.

Memory *shares* entitle a VM to a fraction of physical memory, based on a proportional-share allocation policy. For example, a VM that has twice as many shares as another is generally entitled to consume twice as much memory, subject to their respective *min* and *max* constraints, provided they are both actively using their allocated memory.

6.2 Admission Control

An admission control policy ensures that sufficient unreserved memory and server swap space is available before a VM is allowed to power on. Machine memory must be reserved for the guaranteed *min* size, as well as additional *overhead* memory required for virtualization, for a total of *min* + *overhead*. Overhead memory includes space for the VM graphics frame buffer and various virtualization data structures such as the *pmap* and shadow page tables (see Section 2). Typical VMs reserve 32 MB for overhead, of which 4 to 8 MB is devoted to the frame buffer, and the remainder contains implementation-specific data structures. Additional memory is required for VMs larger than 1 GB.

Disk swap space must be reserved for the remaining VM memory; i.e. *max* − *min*. This reservation ensures the system is able to preserve VM memory under any circumstances; in practice, only a small fraction of this disk space is typically used. Similarly, while memory reservations are used for admission control, actual memory allocations vary dynamically, and unused reservations are not wasted.

6.3 Dynamic Reallocation

ESX Server recomputes memory allocations dynamically in response to various events: changes to system-wide or per-VM allocation parameters by a system administrator, the addition or removal of a VM from the system, and changes in the amount of free memory that cross predefined thresholds. Additional rebalancing is performed periodically to reflect changes in idle memory estimates for each VM.

Most operating systems attempt to maintain a minimum amount of free memory. For example, BSD Unix normally starts reclaiming memory when the percentage of free memory drops below 5% and continues reclaiming until the free memory percentage reaches 7% [18]. ESX Server employs a similar approach, but uses four thresholds to reflect different reclamation states: *high*, *soft*, *hard*, and *low*, which default to 6%, 4%, 2%, and 1% of system memory, respectively.

In the the *high* state, free memory is sufficient and no reclamation is performed. In the *soft* state, the system reclaims memory using ballooning, and resorts to paging only in cases where ballooning is not possible. In the *hard* state, the system relies on paging to forcibly reclaim memory. In the rare event that free memory transiently falls below the *low* threshold, the system continues to reclaim memory via paging, and additionally blocks the execution of all VMs that are above their target allocations.

In all memory reclamation states, the system computes target allocations for VMs to drive the aggregate amount of free space above the *high* threshold. A transition to a lower reclamation state occurs when the amount of free memory drops below the lower threshold. After reclaiming memory, the system transitions back to the next higher state only after significantly exceeding the higher threshold; this hysteresis prevents rapid state fluctuations.

To demonstrate dynamic reallocation we ran a workload consisting of five virtual machines. A pair of VMs executed a Microsoft Exchange benchmark; one VM ran an Exchange Server under Windows 2000 Server, and a second VM ran a load generator client under Windows 2000 Professional. A different pair executed a Citrix MetaFrame benchmark; one VM ran a MetaFrame Server under Windows 2000 Advanced Server, and a second VM ran a load generator client under Windows 2000 Server. A final VM executed database queries with Microsoft SQL Server under Windows 2000 Advanced Server. The Exchange VMs were each configured with 256 MB memory; the other three VMs were each configured with 320 MB. The *min* size for each VM was set to half of its configured *max* size, and memory shares were allocated proportional to the *max* size of each VM.

For this experiment, ESX Server was running on an IBM Netfinity 8500R multiprocessor with eight 550 MHz Pentium III CPUs. To facilitate demonstrating the effects of memory pressure, machine memory was deliberately limited so that only 1 GB was available for executing VMs. The aggregate VM workload was configured to use a total of 1472 MB; with the additional 160 MB required for overhead memory, memory was overcommitted by more than 60%.

Figure 8(a) presents ESX Server allocation states during the experiment. Except for brief transitions early in the run, nearly all time is spent in the *high* and *soft* states. Figure 8(b) plots several allocation metrics over time. When the experiment is started, all five VMs boot concurrently. Windows zeroes the contents of all pages in "physical" memory while booting. This causes the system to become overcommitted almost immediately, as each VM accesses all of its memory. Since the Windows balloon drivers are not started until late in the boot sequence, ESX Server is forced to start paging to disk. Fortunately, the "share before swap" optimization described in Section 4.3 is very effective: 325 MB of zero pages

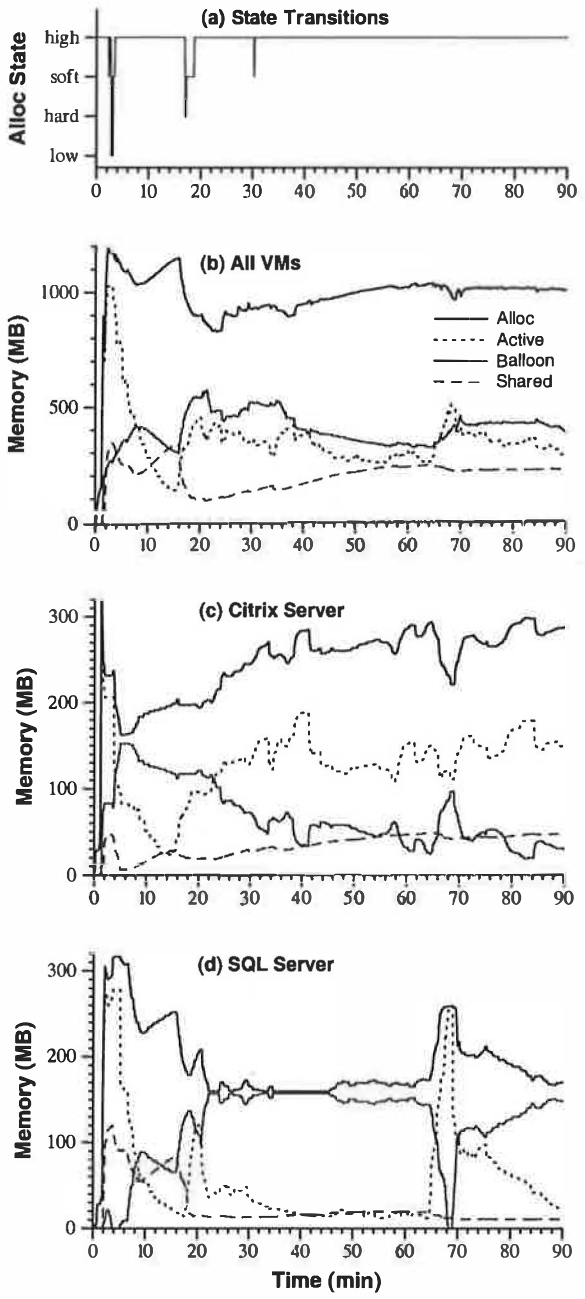


Figure 8: **Dynamic Reallocation.** Memory allocation metrics over time for a consolidated workload consisting of five Windows VMs: Microsoft Exchange (separate server and client load generator VMs), Citrix MetaFrame (separate server and client load generator VMs), and Microsoft SQL Server. (a) ESX Server allocation state transitions. (b) Aggregate allocation metrics summed over all five VMs. (c) Allocation metrics for MetaFrame Server VM. (d) Allocation metrics for SQL Server VM.

are reclaimed via page sharing, while only 35 MB of non-zero data is actually written to disk.[7] As a result of sharing, the aggregate allocation to all VMs approaches 1200 MB, exceeding the total amount of machine memory. Soon after booting, the VMs start executing their application benchmarks, the amount of shared memory drops rapidly, and ESX Server compensates by using ballooning to reclaim memory. Page sharing continues to exploit sharing opportunities over the run, saving approximately 200 MB.

Figures 8(c) and 8(d) show the same memory allocation data for the Citrix MetaFrame Server VM and the Microsoft SQL Server VM, respectively. The MetaFrame Server allocation tracks its active memory usage, and also grows slowly over time as page sharing reduces overall memory pressure. The SQL Server allocation starts high as it processes queries, then drops to 160 MB as it idles, the lower bound imposed by its *min* size. When a long-running query is issued later, its active memory increases rapidly, and memory is quickly reallocated from other VMs.

7 I/O Page Remapping

Modern IA-32 processors support a physical address extension (PAE) mode that allows the hardware to address up to 64 GB of memory with 36-bit addresses [13]. However, many devices that use DMA for I/O transfers can address only a subset of this memory. For example, some network interface cards with 32-bit PCI interfaces can address only the lowest 4 GB of memory.

Some high-end systems provide hardware support that can be used to remap memory for data transfers using a separate I/O MMU. More commonly, support for I/O involving "high" memory above the 4 GB boundary involves copying the data through a temporary *bounce buffer* in "low" memory. Unfortunately, copying can impose significant overhead resulting in increased latency, reduced throughput, or increased CPU load.

This problem is exacerbated by virtualization, since even pages from virtual machines configured with less than 4 GB of "physical" memory may be mapped to machine pages residing in high memory. Fortunately, this same level of indirection in the virtualized memory system can be exploited to transparently remap guest pages between high and low memory.

[7]This is the peak amount of memory paged to disk over the entire run. To avoid clutter, paging metrics were omitted from the graphs; the amount of data swapped to disk was less than 20 MB for the remainder of the experiment.

ESX Server maintains statistics to track "hot" pages in high memory that are involved in repeated I/O operations. For example, a software cache of physical-to-machine page mappings (PPN-to-MPN) associated with network transmits is augmented to count the number of times each page has been copied. When the count exceeds a specified threshold, the page is transparently remapped into low memory. This scheme has proved very effective with guest operating systems that use a limited number of pages as network buffers. For some network-intensive workloads, the number of pages copied is reduced by several orders of magnitude.

The decision to remap a page into low memory increases the demand for low pages, which may become a scarce resource. It may be desirable to remap some low pages into high memory, in order to free up sufficient low pages for remapping I/O pages that are currently "hot." We are currently exploring various techniques, ranging from simple random replacement to adaptive approaches based on cost-benefit tradeoffs.

8 Related Work

Virtual machines have been used in numerous research projects [3, 7, 8, 9] and commercial products [20, 23] over the past several decades. ESX Server was inspired by recent work on Disco [3] and Cellular Disco [9], which virtualized shared-memory multiprocessor servers to run multiple instances of IRIX.

ESX Server uses many of the same virtualization techniques as other VMware products. One key distinction is that VMware Workstation uses a hosted architecture for maximum portability across diverse desktop systems [23], while ESX Server manages server hardware directly for complete control over resource management and improved I/O performance.

Many of the mechanisms and policies we developed were motivated by the need to run existing commodity operating systems without any modifications. This enables ESX Server to run proprietary operating systems such as Microsoft Windows and standard distributions of open-source systems such as Linux.

Ballooning implicitly coaxes a guest OS into reclaiming memory using its own native page replacement algorithms. It has some similarity to the "self-paging" technique used in the Nemesis system [11], which requires applications to handle their own virtual memory operations, including revocation. However, few applications are capable of making their own page replacement decisions, and applications must be modified to participate in an explicit revocation protocol. In contrast, guest operating systems already implement page replacement algorithms and are oblivious to ballooning details. Since they operate at different levels, ballooning and self-paging could be used together, allowing applications to make their own decisions in response to reclamation requests that originate at a much higher level.

Content-based page sharing was directly influenced by the transparent page sharing work in Disco [3]. However, the content-based approach used in ESX Server avoids the need to modify, hook, or even understand guest OS code. It also exploits many opportunities for sharing missed by both Disco and the standard copy-on-write techniques used in conventional operating systems.

IBM's MXT memory compression technology [27], which achieves substantial memory savings on server workloads, provided additional motivation for page sharing. Although this hardware approach eliminates redundancy at a sub-page granularity, its gains from compression of large zero-filled regions and other patterns can also be achieved via page sharing.

ESX Server exploits the ability to transparently remap "physical" pages for both page sharing and I/O page remapping. Disco employed similar techniques for replication and migration to improve locality and fault containment in NUMA multiprocessors [3, 9]. In general, page remapping is a well-known approach that is commonly used to change virtual-to-physical mappings in systems that do not have an extra level of "virtualized physical" addressing. For example, remapping and page coloring have been used to improve cache performance and isolation [17, 19, 21].

The ESX Server mechanism for working-set estimation is related to earlier uses of page faults to maintain per-page reference bits in software on architectures lacking direct hardware support [2]. However, we combine this technique with a unique statistical sampling approach. Instead of tracking references to all pages individually, an aggregate estimate of idleness is computed by sampling a small subset.

Our allocation algorithm extends previous research on proportional-share allocation of space-shared resources [31, 32]. The introduction of a "tax" on idle memory solves a significant known problem with pure share-based approaches [25], enabling efficient memory utilization while still maintaining share-based isolation. The use of economic metaphors is also related to more explicit market-based approaches designed to facilitate decentralized application-level optimization [12].

9 Conclusions

We have presented the core mechanisms and policies used to manage memory resources in ESX Server [29], a commercially-available product. Our contributions include several novel techniques and algorithms for allocating memory across virtual machines running unmodified commodity operating systems.

A new ballooning technique reclaims memory from a VM by implicitly causing the guest OS to invoke its own memory management routines. An idle memory tax was introduced to solve an open problem in share-based management of space-shared resources, enabling both performance isolation and efficient memory utilization. Idleness is measured via a statistical working set estimator. Content-based transparent page sharing exploits sharing opportunities within and between VMs without any guest OS involvement. Page remapping is also leveraged to reduce I/O copying overheads in large-memory systems. A higher-level dynamic reallocation policy coordinates these diverse techniques to efficiently support virtual machine workloads that overcommit memory.

We are currently exploring a variety of issues related to memory management in virtual machine systems. Transparent page remapping can be exploited to improve locality and fault containment on NUMA hardware [3, 9] and to manage the allocation of cache memory to VMs by controlling page colors [17]. Additional work is focused on higher-level grouping abstractions [30, 31], multi-resource tradeoffs [24, 31], and adaptive feedback-driven workload management techniques [5].

Acknowledgements

Mahesh Patil designed and implemented the ESX Server paging mechanism. Mike Nelson and Kinshuk Govil helped develop the hot I/O page remapping technique. Vikram Makhija and Mahesh Patil provided invaluable assistance with quantitative experiments. We would also like to thank Keith Adams, Ole Agesen, Jennifer Anderson, Ed Bugnion, Andrew Lambeth, Beng-Hong Lim, Tim Mann, Michael Mullany, Paige Parsons, Mendel Rosenblum, Jeff Slusher, John Spragens, Pratap Subrahmanyam, Chandu Thekkath, Satyam Vaghani, Debby Wallach, John Zedlewski, and the anonymous reviewers for their comments and helpful suggestions.

References

[1] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. "Information and Control in Gray-Box Systems," *Proc. Symposium on Operating System Principles*, October 2001.

[2] Ozalp Babaoglu and William Joy. "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits," *Proc. Symposium on Operating System Principles*, December 1981.

[3] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Transactions on Computer Systems*, 15(4), November 1997.

[4] Pei Cao, Edward W. Felten, and Kai Li. "Implementation and Performance of Application-Controlled File Caching," *Proc. Symposium on Operating System Design and Implementation*, November 1994.

[5] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, and Amin M. Vahdat. "Managing Energy and Server Resources in Hosting Centers," *Proc. Symposium on Operating System Principles*, October 2001.

[6] R. J. Creasy. "The Origin of the VM/370 Time-Sharing System," *IBM Journal of Research and Development*, 25(5), September 1981.

[7] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Stephen Clawson. "Microkernels Meet Recursive Virtual Machines," *Proc. Symposium on Operating System Design and Implementation*, October 1996.

[8] Robert P. Goldberg. "Survey of Virtual Machine Research," *IEEE Computer*, 7(6), June 1974.

[9] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. "Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors," *Proc. Symposium on Operating System Principles*, December 1999.

[10] Peter H. Gum. "System/370 Extended Architecture: Facilities for Virtual Machines," *IBM Journal of Research and Development'*, 27(6), November 1983.

[11] Steven M. Hand. "Self-Paging in the Nemesis Operating System," *Proc. Symposium on Operating Systems Design and Implementation*, February 1999.

[12] Kieran Harty and David R. Cheriton. "Application-Controlled Physical Memory using External Page-Cache Management," *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[13] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual. Volumes I, II, and III*, 2001.

[14] Bob Jenkins. "Algorithm Alley," *Dr. Dobbs Journal*, September 1997. Source code available from `http://burtleburtle.net/bob/hash/`

[15] Ted Kaehler. "Virtual Memory for an Object-Oriented Language," *Byte*, August 1981.

[16] Minkyong Kim and Brian Noble. "Mobile Network Estimation," *Proc. Seventh Annual International Conference on Mobile Computing and Networking*, July 2001.

[17] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. "OS-Controlled Cache Predictability for Real-Time Systems," *Proc. Third IEEE Real-Time Technology and Applications Symposium*, June 1997.

[18] Marshall K. McKusick, Keith Bostic, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley, 1996.

[19] Theodore H. Romer, Dennis Lee, Brian N. Bershad and J. Bradley Chen. "Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware," *Proc. Symposium on Operating System Design and Implementation*, November 1994.

[20] L. H. Seawright and R. A. McKinnon. "VM/370: A Study of Multiplicity and Usefulness," *IBM Systems Journal*, 18(1), 1979.

[21] Timothy Sherwood, Brad Calder and Joel S. Emer. "Reducing Cache Misses Using Hardware and Software Page Placement," *Proc. International Conference on Supercomputing*, June 1999.

[22] David A. Solomon and Mark Russinovich. *Inside Microsoft Windows 2000*, Third Ed., Microsoft Press, 2001.

[23] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," *Proc. Usenix Annual Technical Conference*, June 2001.

[24] David G. Sullivan, Robert Haas, and Margo I. Seltzer. "Tickets and Currencies Revisited: Extensions to Multi-Resource Lottery Scheduling," *Proc. Seventh Workshop on Hot Topics in Operating Systems*, March 1999.

[25] David G. Sullivan and Margo I. Seltzer. "Isolation with Flexibility: A Resource Management Framework for Central Servers," *Proc. Usenix Annual Technical Conference*, June 2000.

[26] Andrew S. Tanenbaum. *Modern Operating Systems*, Prentice Hall, 1992.

[27] R. Tremaine, P. Franaszek, J. Robinson, C. Schulz, T. Smith, M. Wazlowski, and P. Bland. "IBM Memory Expansion Technology (MXT)," *IBM Journal of Research and Development'*, 45(2), March 2001.

[28] Andrew Tridgell. "dbench" benchmark. Available from `ftp://samba.org/pub/tridge/dbench/`, September 2001.

[29] VMware, Inc. *VMware ESX Server User's Manual Version 1.5*, Palo Alto, CA, April 2002.

[30] Carl A. Waldspurger and William E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. Symposium on Operating System Design and Implementation*, November 1994.

[31] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, Ph.D. thesis, Technical Report MIT/LCS/TR-667, September 1995.

[32] Carl A. Waldspurger and William E. Weihl. "An Object-Oriented Framework for Modular Resource Management," *Proc. Fifth Workshop on Object-Orientation in Operating Systems*, October 1996.

Scale and Performance in the Denali Isolation Kernel

Andrew Whitaker, Marianne Shaw, and Steven D. Gribble
University of Washington
{andrew,mar,gribble}@cs.washington.edu

Abstract

This paper describes the Denali isolation kernel, an operating system architecture that safely multiplexes a large number of untrusted Internet services on shared hardware. Denali's goal is to allow new Internet services to be "pushed" into third party infrastructure, relieving Internet service authors from the burden of acquiring and maintaining physical infrastructure. Our isolation kernel exposes a virtual machine abstraction, but unlike conventional virtual machine monitors, Denali does not attempt to emulate the underlying physical architecture precisely, and instead modifies the virtual architecture to gain scale, performance, and simplicity of implementation. In this paper, we first discuss design principles of isolation kernels, and then we describe the design and implementation of Denali. Following this, we present a detailed evaluation of Denali, demonstrating that the overhead of virtualization is small, that our architectural choices are warranted, and that we can successfully scale to more than 10,000 virtual machines on commodity hardware.

1 Introduction

Advances in networking and computing technology have accelerated the proliferation of Internet services, an application model in which service code executes in the Internet infrastructure rather than on client PCs. Many applications fit this model, including web sites, search engines, and wide area platforms such as content distribution networks, caching systems, and network experimentation testbeds [25]. The Denali project seeks to encourage and enhance the Internet service model by making it possible for untrusted software services to be "pushed" safely into third party hosting infrastructure, thereby separating the deployment of services from the management of the physical infrastructure on which they run.

While this has clear benefits, it also faces difficult technical challenges. One challenge is *scale*: for cost-efficiency and convenience, infrastructure providers will need to multiplex many services on each server machine, as it would be prohibitively expensive to dedicate a separate machine to each service. A second challenge is *security*: infrastructure providers cannot trust hosted services, and services will not trust each other. There must be strong isolation between services, both for security and to enforce fair resource provisioning.

In this paper, we present the design, implementation, and evaluation of the Denali *isolation kernel*, an x86-based operating system that isolates untrusted software services in separate protection domains. The architecture of Denali is similar to that of virtual machine monitors such as Disco [6], VMWare [31], and VM/370 [9]. A virtual machine monitor carves a physical machine into multiple virtual machines; by virtualizing all hardware resources, a VMM can prevent one VM from even naming the resources of another VM, let alone modifying them.

To support unmodified legacy "guest" OSs and applications, conventional VMMs have the burden of faithfully emulating the complete architecture of the physical machine. However, modern physical architectures were not designed with virtualization or scale in mind. In Denali, we have reconsidered the exposed virtual architecture, making substantial changes to the underlying physical architecture to enhance scalability, performance, and simplicity, while retaining the strong isolation properties of VMMs.

For example, although Denali exposes virtual disks and NICs, their interfaces have been redesigned for simplicity and performance. Similarly, Denali exposes an instruction set architecture which is similar to x86 (to gain the performance benefits of directly executing instructions on the host processor), but in which non-virtualizable aspects have been hidden for simplicity, and in which the interrupt model has been changed for scalability.

The cost of Denali's virtual architecture modifications is backwards compatibility: Denali is not able to run unmodified legacy guest operating systems. However, the Denali virtual architecture is complete, in the sense that a legacy operating system could be ported to Denali (although this is still work in progress). To evaluate Denali in the absence of a ported legacy OS, we implemented our own lightweight guest OS, called Ilwaco, which contains a port of the BSD TCP/IP networking stack, thread support, and support for a subset of the POSIX API. We have ported several applications to Ilwaco, including a web server, the Quake II game server, telnet, and various utilities.

1.1 Contributions

The contributions of this paper are:

1. A case for isolation kernels, an OS structure for isolating untrusted software services;

2. A set of design principles for isolation kernels, arguing for a VMM-like structure, but with strategic modifications to the virtual architecture for scalability, performance, and simplicity;

3. The design, implementation, and evaluation of the Denali isolation kernel, focusing on the challenges of scale, and demonstrating that Denali can scale to over 10,000 VMs on commodity hardware.

The rest of this paper is organized as follows. In Section 2, we describe the various classes of applications we hope to enable, and derive design principles of isolation kernels. Section 3 discusses the design and implementation of the Denali isolation kernel. In Section 4, we evaluate our implementation, focusing on issues of scale. We compare Denali to related work in Section 5, and we conclude in Section 6.

2 The Case for Isolation Kernels

Many applications and services would benefit from the ability to push untrusted code into the Internet infrastructure. We outline some of these below, and use them to motivate the properties required by an isolation kernel.

Supporting dynamic content in content delivery systems: a progression of content delivery systems has been introduced in recent years, including CDNs, proxy caches [34], and peer-to-peer networks [30]. All suffer from the limitation that only static content is supported, whereas a large and increasing fraction of content is dynamically generated [34]. Dynamic content distribution requires the ability to execute and isolate untrusted content generation code.

Pushing Internet services into virtual hosting infrastructure: a "virtual" hosting center would allow new Internet services to be uploaded into managed data centers. In addition to supporting commercial services, we believe virtual hosting centers would encourage the emergence of a grassroots development community for Internet services, similar to the shareware community that exists for desktop applications.

Internet measurement and experimentation infrastructure: NIMI [25] and CAIRN [2] have sought to deploy wide-area testbeds to support network measurement research. Recent projects such as Chord [30] would benefit from the ability to deploy research prototypes at scale across the Internet. Whether for measurement or prototyping, the infrastructure must be able to multiplex and isolate mutually distrusting experiments.

Mobile code: deploying mobile code in routers and servers has been proposed by both active networks and mobile agent systems [19].

All of these services and applications share several properties. For the sake of cost-efficiency, multiple services will need to be multiplexed on shared infrastructure. As a result, software infrastructure must exist to isolate multiplexed services from each other: a service must not be able to corrupt another service or the underlying protection system. Additionally, performance isolation is required to bound each service's resource consumption. Finally, the degree of information sharing between these multiplexed services will be small, or entirely non-existent. Because of this, it is reasonable to strengthen isolation at the cost of high sharing overhead.

As we will argue in detail, no existing software system has the correct set of properties to support this emerging class of Internet services. Existing software protection systems (including operating systems, language-based protection techniques, and virtual machine monitors) suffer from some combination of security vulnerabilities, complexity, insufficient scalability, poor performance, or resource management difficulties. We believe that a new software architecture called an *isolation kernel* is required to address the challenges of hosting untrusted services.

2.1 Isolation Kernel Design Principles

An isolation kernel is a small-kernel operating system architecture targeted at hosting multiple untrusted applications that require little data sharing. We have formulated four principles that govern the design of isolation kernels.

1. Expose low-level resources rather than high-level abstractions. In theory, one might hope to achieve isolation on a conventional OS by confining each untrusted service to its own process (or process group). However, OSs have proven ineffective at containing insecure code, let alone untrusted or malicious services. An OS exposes high-level abstractions, such as files and sockets, as opposed to low-level resources such as disk blocks and network packets. High-level abstractions entail significant complexity and typically have a wide API, violating the security principle of economy of mechanism [29]. They also invite "layer below" attacks, in which an attacker gains unauthorized access to a resource by requesting it below the layer of enforcement [18].

An isolation kernel exposes hardware-level resources, displacing the burden of implementing operating systems abstractions to user-level code. In this respect, an isolation kernel resembles other "small kernel" architectures such as microkernels [1], virtual machine monitors [6], and Exokernels [20]. Although small kernel architectures were once viewed as prohibitively inefficient, modern hardware improvements have made performance less of a concern.

2. Prevent direct sharing by exposing only private, virtualized namespaces. Conventional OSs facilitate protected data sharing between users and applications by exposing global namespaces, such as file systems and shared memory regions. The presence of these sharing mechanisms introduces the problem of specifying a complex access control policy to protect these globally exposed resources.

Little direct sharing is needed across Internet services, and therefore an isolation kernel should prevent direct sharing by confining each application to a private namespace. Memory pages, disk blocks, and all other resources should be virtualized, eliminating the need for a complex access control policy: the only sharing allowed is through the virtual network.

Both principles 1 and 2 are required to achieve strong isolation. For example, the UNIX `chroot` command discourages direct sharing by confining applications to a private file system name space. However, because `chroot` is built on top of the file system abstraction, it has been compromised by a layer-below attack in which the attacker uses a cached file descriptor to subvert file system access control.

Although our discussion has focused on security isolation, high-level abstractions and direct sharing also reduce performance isolation. High-level abstractions create contention points where applications compete for resources and synchronization primitives. This leads to the effect of "crosstalk" [23], where application resource management decisions interfere with each other. The presence of data sharing leads to hidden shared resources like the file system buffer cache, which complicate precise resource accounting.

3. Zipf's Law implies the need for scale. An isolation kernel must be designed to scale up to a large number of services. For example, to support dynamic content in web caches and CDNs, each cache or CDN node will need to store content from hundreds (if not thousands) of dynamic web sites. Similarly, a wide-area research testbed to simulate systems such as peer-to-peer content sharing applications must scale to millions of simulated nodes. A testbed with thousands of contributing sites would need to support thousands of virtual nodes per site.

Studies of web documents, DNS names, and other network services show that popularity tends to be driven by Zipf distributions [5]. Accordingly, we anticipate that isolation kernels must be able to handle Zipf workloads. Zipf distributions have two defining traits: most requests go to a small set of popular services, but a significant fraction of requests go to a large set of unpopular services. Unpopular services are accessed infrequently, reinforcing the need to multiplex many services on a single machine.

To scale, an isolation kernel must employ techniques to minimize the memory footprint of each service, including metadata maintained by the kernel. Since the set of all unpopular services won't fit in memory, the kernel must treat memory as a cache of popular services, swapping inactive services to disk. Zipf distributions have a poor cache hit rate [5], implying that we need rapid swapping to reduce the cache miss penalty of touching disk.

4. Modify the virtualized architecture for simplicity, scale, and performance. Virtual machine monitors (VMMs), such as Disco [6] and VM/370 [9], adhere to our first two principles. These systems also strive to support legacy OSs by precisely emulating the underlying hardware architecture. In our view, the two goals of isolation and hardware emulation are orthogonal. Isolation kernels decouple these goals by allowing the virtual architecture to deviate from the underlying physical architecture. By so doing, we can enhance properties such as performance, simplicity, and scalability, while achieving the strong isolation that VMMs provide.

The drawback of this approach is that it gives up support for unmodified legacy operating systems. We have chosen to focus on the systems issues of scalability and performance rather than backwards compatibility for legacy OSs. However, we are currently implementing a port of the Linux operating system to the Denali virtual architecture; this port is still work in progress.

3 The Denali Isolation Kernel

The Denali isolation kernel embodies all of the principles described in the previous section of this paper. Architecturally, Denali is a thin software layer that runs directly on x86 hardware. Denali exposes a virtual machine (VM) architecture that is based on the x86 hardware, and supports the secure multiplexing of many VMs on an underlying physical machine. Each VM can run its own "guest" operating system and applications (Figure 1).

This section of the paper presents the design of the Denali virtual architecture, and the implementation of an isolation kernel to support it. We also describe the *Ilwaco* guest OS, which is tailored for building Internet services that execute on the Denali virtual architecture.

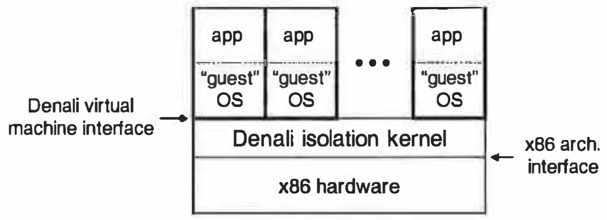


Figure 1: **The Denali architecture:** the Denali isolation kernel is a thin software layer that exposes a virtual machine abstraction that is based on the underlying x86 architecture.

3.1 The Denali Virtual Architecture

The Denali virtual architecture consists of an instruction set, a memory architecture, and an I/O architecture (including an interrupt model). We describe each of these components in turn.

3.1.1 ISA

The Denali virtual instruction set was designed for both performance and simplicity. The ISA primarily consists of a subset of the x86 instruction set, so that most virtual instructions execute directly on the physical processor. The x86 ISA is not strictly virtualizable, as it contains instructions that behave differently in user mode and kernel mode [17, 27]; x86 virtual machine monitors must use a combination of complex binary rewriting and memory protection techniques to virtualize these instructions. Since Denali is not designed to support legacy OSs, our virtual architecture simply defines these instructions to have ambiguous semantics. If a VM executes one of these instructions, at worst the VM could harm itself. In practice, they are rarely used; most deal with legacy architecture features like segmentation, and none are emitted by C compilers such as gcc (unless they appear in inlined assembly fragments).

Denali defines two purely virtual instructions. The first is an "idle-with-timeout" instruction that helps VMs avoid wasting their share of the physical CPU by executing OS idle loops. The idle-with-timeout instruction lets a VM halt its virtual CPU for either a bounded amount of physical time, or until an interrupt arrives for the VM.[1] The second purely virtual instruction simply allows a virtual machine to terminate its own execution.

Denali adds several virtual registers to the x86 register file, to expose system information such as CPU speed, the size of memory, and the current system time. Virtual registers also provide a lightweight communication mechanism between virtual machines and the kernel. For example, we implemented Denali's interrupt-enabled flag as a virtual register.

[1]Denali's idle instruction is similar to the x86 halt instruction, which is executed to put the system into a low-power state during idle periods. Denali's timeout feature allows for fine-grained CPU sharing.

3.1.2 Memory Architecture

Each Denali VM is given its own (virtualized) physical 32-bit address space. A VM may only access a subset of this 32-bit address space, the size and range of which is chosen by the isolation kernel when the VM is instantiated. The kernel itself is mapped into a portion of the address space that the VM cannot access; because of this, we can avoid physical TLB flushes on VM/VMM crossings.

By default, a VM cannot virtualize its own (virtualized) physical address space: in other words, by default, there is no virtual MMU. In this configuration, a VM's OS shares its address space with applications, similar to a libOS in Exokernel [20]. Exposing a single address space to each VM improves performance, by avoiding TLB flushes during context switches between applications in the same VM [31].

We have recently added support for an optional, virtual MMU to Denali; this virtual MMU looks nothing like the underlying x86-based physical MMU, but instead is modeled after a simple software-loaded TLB, similar to those of modern RISC architectures. A software-loaded TLB has the advantage that the VM itself gets to define its own page-table structure, and the software TLB interface between the VMM and the VM is substantially simpler than the more complex page table interface mandated by the x86 hardware-loaded TLB architecture.

3.1.3 I/O Devices and Interrupt Model

Denali exposes several virtual I/O devices, including an Ethernet NIC, a disk, a keyboard, a console, and a timer. Denali's virtual devices have similar functionality to their physical counterparts, but they expose a simpler interface. Physical devices often have "chatty" interfaces, requiring many programmed I/O instructions per operation. VMMs that emulate real hardware devices suffer high overhead, since each PIO must be emulated [31]. Another benefit of simplification is portability: our virtual device interfaces are independent of the underlying physical devices.

Denali has chosen to omit many x86 architectural features. Virtual devices have been simplified to begin in a well-known, pre-initialized state when a VM boots. This simplifies both the Denali isolation kernel and guest OSs by eliminating the need to probe virtual devices on boot. Denali also does not expose the BIOS[2] or segmentation hardware, because these features are rarely used outside of system boot.

[2]The BIOS is also involved in power management; Denali does not expose this to VMs.

Denali exposes virtual interrupts to VMs, much in the same way that the physical x86 architecture exposes real interrupts to the host processor. Virtual interrupts are typically triggered by physical interrupts, such as when an Ethernet packet arrives that is destined for a particular VM. However, not all physical interrupts cause virtual interrupts; for example, a packet may arrive that is not destined for any of the running VMs, in which case the isolation kernel simply drops the packet without raising a virtual interrupt.

Denali's interrupt dispatch model differs significantly from the underlying x86 hardware to better support the multiplexing of many virtual machines. As the number of simultaneously running VMs grows, it becomes increasingly unlikely that the VM which is the ultimate recipient of a physical interrupt is executing when the interrupt is raised. In some cases, the target VM could even be swapped out to disk. Rather than preserving the immediate interrupt semantics of x86, Denali delays and batches interrupts destined for non-running VMs. A VM receives pending interrupts once it begins its normal scheduler quantum, and if multiple interrupts are pending for a VM, all interrupts are delivered in a single VMM/VM crossing. This is similar to the Mach 3.0 user-level I/O interface [15].

Denali's asynchronous, batched interrupt model changes the semantics of timing-related interrupts. For example, a conventional timer interrupt implies that a fixed-size time interval has just passed. In Denali, a virtual timer interrupt implies that some amount of physical time has passed, but the duration may depend on how many other VMs are contending for the CPU. As a result, the interpretation of timer interrupts in the implementation of guest OS software timers must be altered.

3.2 Isolation Kernel Implementation

The Denali isolation kernel runs directly on x86 hardware. The core of the kernel, including multiprogramming, paging, and virtual device emulation, was implemented from scratch; we used the Flux OSKit [14] for device drivers and other hardware support routines, and some support libraries such as libc.

The isolation kernel serves two roles: it implements the Denali virtual architecture, and it multiplexes physical resources across competing VMs. We have maintained a strict separation between resource management policy and mechanism, so that we could implement different policies without affecting other aspects of the isolation kernel.

3.2.1 CPU Virtualization

Denali uses standard multiprogramming techniques to multiplex the CPU across VMs. The isolation kernel maintains a per-VM thread structure, which contains a kernel stack, space for the register file, and the thread status. The policy for multiplexing the CPU is split into two components: a *gatekeeper policy* enforces admission control by choosing a subset of active machines to admit into the system; the rest are swapped to disk, as we will describe later. The *scheduler policy* controls context switching among the set of admitted machines.

The gatekeeper admits machines in FIFO order as long as there are a minimum number of physical backing pages available. The scheduler uses round-robin scheduling among the set of admitted machines. These policies were chosen because they are simple and starvation-free. When a VM issues an idle-with-timeout instruction, it is removed from scheduler consideration until its timer fires, or a virtual interrupt arrives. As compensation for idling, a VM receives higher scheduler priority for its next quantum.

Virtual registers are stored in a page at the beginning of a VM's (virtual) physical address space. This page is shared between the VM and the isolation kernel, avoiding the overhead of kernel traps for register modifications. In other respects, the virtual registers behave like normal memory (for example, they can be paged out to disk).

Because Denali's ISA is based on x86, we can use existing x86 compilers and linkers when authoring OS or application code to run in a Denali VM. In particular, we have been primarily using the `gcc` C compiler and the `ld` linker on Linux, although we did need to change the link map used by `ld` to take Denali's memory architecture into account.

3.2.2 Memory Management

The (virtual) physical address space exposed to a VM has two components: a portion that is accessible to the VM, and a protected portion accessible only to the isolation kernel. Each VM also has a *swap region* allocated on behalf of it by the isolation kernel; this swap region is striped across local disks. The swap region is used by the isolation kernel to swap or page out portions of the VM's address space. Swap regions are statically allocated at VM creation time, and are large enough to hold the entire VM-visible address space. Static allocation drastically reduces the amount of bookkeeping metadata in the isolation kernel: each swap region is completely described by 20 bytes of kernel memory. Static allocation wastes disk capacity in return for performance and scalability, but the decreasing cost of storage capacity makes this trade-off worthwhile.

The isolation kernel is pinned in physical memory, but VMs are paged in on demand. Upon taking a page fault, the kernel verifies that the faulting VM hasn't accessed an illegal virtual address, allocates necessary page tables, and initiates a read from the VM's swap region.

The system periodically redistributes physical memory from inactive VMs to active VMs. We use the WSClock [7] page replacement algorithm, which attempts to maintain each VM's working set in memory by maintaining a virtual time stamp along with a clock reference bit. This helps reduce thrashing, and is more fair to machines that experience heavy paging (such as reactivated machines that are entirely swapped out). To encourage good disk locality, all memory buffers for a given VM are clustered together in the clock circular list.

For the remainder of this paper, we configured the system to expose only 16MB of accessible (virtual) physical address space to each VM. This models the challenging scenario of having many small services multiplexed on the same hardware. Because virtual MMUs are such a recent addition and are still being performance optimized, we did not turn on virtual MMU support for the experiments presented in Section 4. Although we hope that enabling virtual MMU support will not affect our overall performance results, we have not yet demonstrated this.

3.2.3 I/O Devices and Interrupt Model

Denali emulates a switched Ethernet LAN connecting all VMs. Each VM is assigned a virtual Ethernet NIC; from the perspective of external physical hosts, it appears as though each VM has its own physical Ethernet card. VMs interact with the virtual NIC using standard programmed I/O instructions, although the interface to the virtual NIC is drastically simpler than physical NICs, consisting only of a packet send and a packet receive operation. On the reception path, the isolation kernel emulates an Ethernet switch by demultiplexing incoming packets into a receive queue for the destination virtual machine. VMs can only process packets during their scheduler quantum, effectively implementing a form of lazy-receiver processing [11]. On the transmit path, the kernel maintains a per-machine queue of outbound packets which a VM can fill during its scheduler quantum. These transmit queues are drained according to a packet scheduler policy; Denali currently processes packets in round-robin order from the set of actively sending VMs.

Denali provides virtual disk support to VMs. The isolation kernel contains a simple file system in which it manages persistent, fixed-sized virtual disks. When a VM is instantiated, the isolation kernel exposes a set of virtual disks to it; the VM can initiate asynchronous reads and writes of 4KB blocks from the disks to which it has been given access. Because virtual disks exist independently of virtual machines, Denali trivially supports optimizations such as the read-only sharing of virtual disks across VMs. For example, if multiple VMs all use the same kernel boot image, that boot image can be stored on a single read-only virtual disk and shared by the VMs.

Denali also emulates a keyboard, console, and timer devices. These virtual devices do not differ significantly from physical hardware devices, and we do not describe them further.

Denali's batched interrupt model is implemented by maintaining a bitmask of pending interrupts for each VM. When a virtual interrupt arrives, the kernel posts the interrupt to the bitmask, activates the VM if it is idle, and clears any pending timeouts. When a VM begins its next quantum, the kernel uploads the bitmask to a virtual register, and transfers control to an interrupt handler. A VM can disable virtual interrupts by setting a virtual register value; VMs can never directly disable physical interrupts.

3.2.4 Supervisor Virtual Machine

Denali gives special privileges to a *supervisor VM*, including the ability to create and destroy other VMs. Because complexity is a source of security vulnerabilities, wherever possible we have displaced complexity from the isolation kernel to the supervisor VM. For example, the isolation kernel does not have a network stack: if a remote VM image needs to be downloaded for execution, this is done by the supervisor VM. Similarly, the supervisor VM keeps track of the association between virtual disks and VMs, and is responsible for initializing or loading initial disk images into virtual disks. The supervisor VM can be accessed via the console, or through a simple telnet interface. In a production system, the security of the supervisor VM should be enhanced by using a secure login protocol such as ssh.

3.3 Ilwaco Guest OS

Although the Denali virtual machine interface is functionally complete, it is not a convenient interface for developing applications. Accordingly, we have developed the Ilwaco guest operating system which presents customary high-level abstractions. Ilwaco is implemented as a library, in much the same fashion as a Exokernel libOS. Applications directly link against the OS; there is no hardware-enforced protection boundary.

Ilwaco contains the Alpine user-level TCP stack [12], a port of the FreeBSD 3.3 stack. We modified Alpine to utilize Denali's virtual interrupt and timer mechanisms, and linked the stack against a device driver for the Denali virtual Ethernet NIC.

Ilwaco contains a thread package that supports typical thread primitives, locks, and condition variables. If there are no runnable threads, the thread scheduler invokes the idle-with-timeout virtual instruction to yield the CPU. Ilwaco also contains a subset of libc, including basic console I/O, string routines, pseudo-random number generation, and memory management. Most of these routines were ported from OSKit libraries; some functions needed to be modified to interact with Denali's virtual hardware. For example, `malloc` reads the size of (virtual) physical memory from a virtual register.

4 Evaluation

This section presents a quantitative evaluation of the Denali isolation kernel. We ran microbenchmarks to (1) quantify the performance of Denali's primitive operations, (2) validate our claim that our virtual architecture modifications result in enhanced scale, performance, and simplicity, and (3) characterize how our system performs at scale, and why. As previously mentioned, none of these experiments were run with the virtual MMU enabled. Additionally, in all experiments, our VMs ran with their data in virtual core, and as such, they did not exercise the virtual disks.[3]

In our experiments, Denali ran on a 1700MHz Pentium 4 with 256KB of L2 cache, 1GB of RAM, an Intel PRO/1000 PCI gigabit Ethernet card connected to an Intel 470T Ethernet switch, and three 80 GB 7200 RPM Maxtor DiamondMax Plus IDE drives with 2 MB of buffering each. For any experiment involving the network, we used a 1500 byte MTU. To generate workloads for network benchmarks, we used a mixture of 1700MHz Pentium 4 and 930MHz Pentium III machines.

4.1 Basic System Performance

To characterize Denali's performance, we measured the context switching overhead between VMs, and the swap disk subsystem performance. We also characterized virtualization overhead by analyzing packet dispatch latency, and by comparing the application-level TCP and HTTP throughput of Denali with that of BSD.

[3]Of course, our scaling experiments do stress the swapping functionality in the isolation kernel itself.

4.1.1 VM Context Switching Overhead

To measure context-switching overhead, we considered two workloads: a "worst-case" that cycles through a large memory buffer between switches, and a "best-case" that does not touch memory between switches. For the worst-case workload, context switch time starts at 3.9 μs for a single virtual machine, and increases to 9 μs for two or more VMs. For the best-case workload, the context switch time starts at 1.4 μs for a single virtual machine, and it increases slightly as the number of VMs increases; the slight increases coincide with the points at which the capacity of the L1 and L2 caches become exhausted. These results are commensurate with process context switching overheads in modern OSs.

4.1.2 Swap Disk Microbenchmarks

Denali stripes VM swap regions across physical disks. To better understand factors that influence swap performance at scale, we benchmarked Denali's disk latency and throughput for up to three attached physical disks. The results are presented in Table 1; all measured throughputs and latencies were observed to be limited by the performance of the physical disks, but not the Denali isolation kernel. For three disks, a shared PCI bus became the bottleneck, limiting sequential throughput.

4.1.3 Packet Dispatch Latency

Figure 2 shows packet processing costs for application-level UDP packets, for both 100 and 1400 byte packets. A transmitted packet first traverses the Alpine TCP/IP stack and then is processed by the guest OS's Ethernet device driver. This driver signals the virtual NIC using a PIO, resulting in a trap to the isolation kernel. Inside the kernel, the virtual NIC implementation copies the packet out of the guest OS into a transmit FIFO. Once the network scheduler has decided to transmit the packet, the physical device driver is invoked. Packet reception essentially follows the same path in reverse.

On the transmission path, our measurement ends when the physical device driver signals to the NIC that a new packet is ready for transmission; packet transmission costs therefore do not include the time it takes the packet to be DMA'ed into the NIC, the time it takes the NIC to transmit the packet on the wire, or the interrupt that the NIC generates to indicate that the packet has been transmitted successfully. On the reception path, our measurement starts when a physical interrupt arrives from the NIC; packet reception costs therefore include interrupt processing and interacting with the PIC.

The physical device driver and VM's TCP/IP stack incur significantly more cost than the isolation

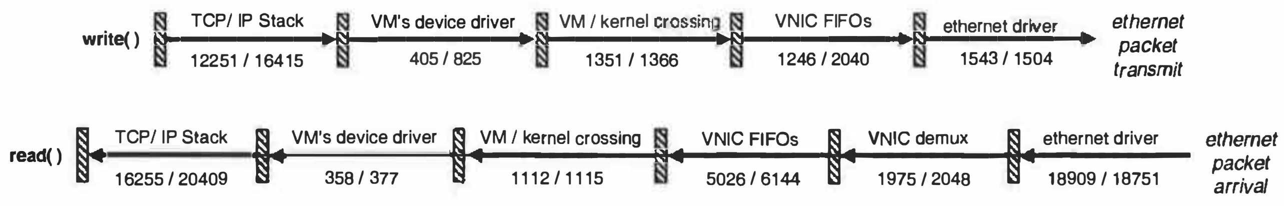


Figure 2: **Packet processing overhead:** these two timelines illustrate the cost (in cycles) of processing a packet, broken down across various functional stages, for both packet reception and packet transmission. Each pair of numbers represents the number of cycles executed in that stage for 100 byte and 1400 byte packets, respectively.

	latency	random throughput	sequential throughput
1 disk	7.1 / 5.9	2.20 / 2.66	38.2 / 31.5
2 disks	7.0 / 5.8	4.45 / 5.41	75.6 / 63.5
3 disks	7.0 / 5.8	6.71 / 8.10	91.3 / 67.1

Table 1: **Swap disk microbenchmarks:** latency (ms), random throughput (MB/s), and sequential throughput (MB/s) versus the number of disks. Numbers separated by a slash are for reads and writes, respectively.

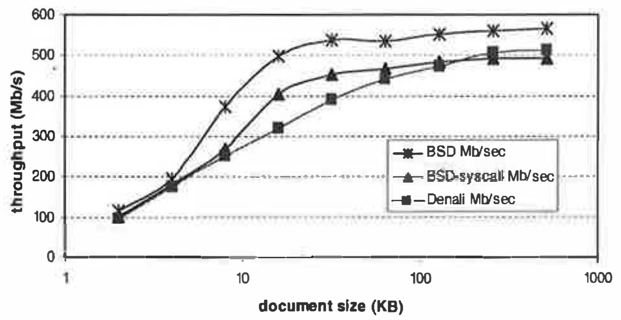


Figure 3: **Comparing web server performance on Denali and BSD:** performance is comparable, confirming that virtualization overhead is low. The "BSD-syscall" line corresponds to a version of the BSD web server in which an extra system call was added per packet, to approximate user-level packet delivery in Denali.

kernel, confirming that the cost of network virtualization is low. The physical driver consumes 43.3% and 38.4% of the total packet reception costs for small and large packets, respectively. Much of this cost is due to the Flux OSKit's interaction with the 8259A PIC; we plan on modifying the OSKit to use the more efficient APIC in the future. The TCP stack consumes 37.3% and 41.8% of a small and large packet processing time, respectively.

The transmit path incurs two packet copies and one VM/kernel boundary crossing; it may be possible to eliminate these copies using copy-on-write techniques. The receive path incurs the cost of a packet copy, a buffer deallocation in the kernel, and a VM/kernel crossing. The buffer deallocation procedure attempts to coalesce memory back into a global pool and is therefore fairly costly; with additional optimization, we believe we could eliminate this.

4.1.4 TCP and HTTP Throughput

As a second measurement of networking performance on Denali, we compared the TCP-level throughput of BSD and a Denali VM running Ilwaco. To do this, we compiled a benchmark application on both Denali and BSD, and had each application run a TCP throughput test to a remote machine. We configured the TCP stacks in all machines to use large socket buffers. The BSD-Linux connection was able to attain a maximum throughput of 607 Mb/s, while Denali-Linux achieved 569 Mb/s, a difference of 5%.

As further evaluation, we measured the performance of a single web server VM running on Denali. Our home-grown web server serves static content out of (virtual) physical memory. For comparison, we ported our web server to BSD by compiling and linking the unmodified source code against a BSD library implementation of the Ilwaco system call API. Figure 3 shows the results.

Denali's application-level performance closely tracks that of BSD, although for medium-sized documents (50-100KB), BSD outperforms Denali by up to 40%. This difference in performance is due to the fact that Denali's TCP/IP stack runs at the user-level, implying that all network packets must cross the user/kernel boundary. In contrast, in BSD, most packets are handled by the kernel, and only data destined for the application crosses the user-kernel boundary. A countervailing force is system calls: in Denali, system calls are handled within the user-level by the Ilwaco guest OS; in BSD, system calls must cross the user-kernel boundary.

For small documents, there are about as many system calls per connection in BSD (`accept`, `reads`, `writes`, and `close`) as there are user/kernel packet crossings in Denali. For large documents, the system bottleneck becomes the Intel PRO/1000 Ethernet card. Therefore, it is only for medium-sized documents that the packet delivery to the user-level networking stack in Denali induces a noticeable penalty; we confirmed this effect by adding a system call per packet to the BSD web server, observing that with this additional overhead, the BSD performance

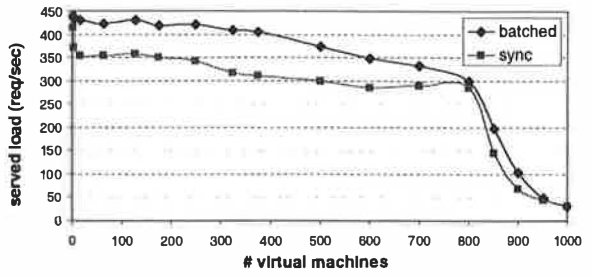


Figure 4: **Benefits of batched, asynchronous interrupts:** Denali's interrupt model leads to a 30% performance improvement in the web server when compared to synchronous interrupts, but at large scale (over 800 VMs), paging costs dominate.

closely matched that of Denali even for medium-sized documents (Figure 3).

4.2 Scale, Performance, and Simplicity and the Denali Virtual Architecture

The virtual architecture exposed by Denali was designed to enhance scalability, performance, and simplicity. In this section, we provide quantitative evidence to back these claims. Specifically, we demonstrate that batched asynchronous interrupts have performance and scalability benefits, that Denali's idle-with-timeout instruction is crucial for scalability, that Denali's simplified virtual NIC has performance advantages over an emulated real NIC, and that the source code complexity of Denali is substantially less than that of even a minimal Linux kernel.

4.2.1 Batched, Asynchronous Interrupts

Denali utilizes a batched, asynchronous model for virtual interrupt delivery. In Figure 4, we quantify the performance gain of Denali's batched, asynchronous interrupt model, relative to the performance of synchronous interrupts. To gather the synchronous interrupt data, we modified Denali's scheduler to immediately context switch into a VM when an interrupt arrives for it. We then measured the aggregate performance of our web server application serving a 100KB document, as a function of the number of simultaneously running VMs. For a small number of VMs, there was no apparent benefit, but up to a 30% gain was achieved with batched interrupts for up to 800 VMs. Most of this gain is attributable to a reduction in context switching frequency (and therefore overhead). For a very large number of VMs (over 800), performance was dominated by the costs of the isolation kernel paging VMs in and out of core.

4.2.2 Idle-with-timeout Instruction

To measure the benefit of the idle-with-timeout virtual instruction, we compared the performance of

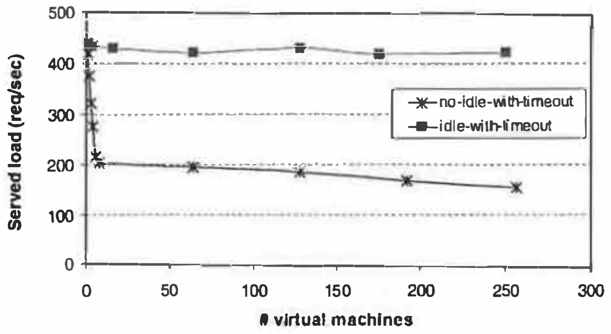


Figure 5: **Idle-with-timeout benefits:** idle-with-timeout leads to higher performance at scale, when compared to an idle instruction with no timeout feature.

web server VMs serving 100KB documents across in two scenarios. In the first scenario, the VMs exploited the timeout feature: a guest OS with no schedulable threads invokes the idle-with-timeout instruction with a timeout value set to the smallest pending TCP retransmission timer. In the second scenario, VMs did not use the timeout feature, idling only when there were no schedulable threads and no pending TCP retransmission timers.

The performance difference was substantial: Figure 5 shows that as the number of VMs scales up, overall performance drops by more than a factor of two without the timeout feature. The precipitous drop in performance for small numbers of VMs happens because the entire offered load is focused on those few VMs, ensuring that all of them have active connections; an active connection means that retransmission timers are likely pending, preventing the VM from idling. As the number of VMs grows, the same aggregate workload is spread over more VMs, meaning that any individual VM is less likely to have active connections preventing it from idling. This results in an easing off of additional overhead as the system scales.

In general, a timeout serves as a hint to the isolation kernel; without this hint, the kernel cannot determine whether any software timers inside a VM are pending, and hence will not know when to reschedule the VM. As a result, without the timeout feature, a VM has no choice but to spin inside its idle loop to ensure that any pending software timers fire.

4.2.3 Simplified Virtual Ethernet

Denali's virtual Ethernet has been streamlined for simplicity and performance. Real hardware network adapters often require multiple programmed I/O instructions to transmit or receive a single packet. For example, the Linux pcnet32 driver used by VMWare workstation [31] issues 10 PIOs to receive a packet and 12 PIOs to transmit a packet.

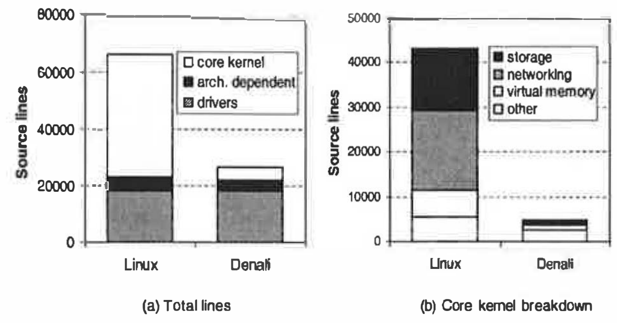


Figure 6: **Source code complexity:** number of source lines in Linux 2.4.16 and Denali. Denali is roughly half the size of Linux in total source lines. Denali's core kernel (without device drivers and platform-dependent code) is an order-of-magnitude smaller than Linux.

VMMs which support unmodified legacy OSs must trap and emulate these PIOs, resulting in additional overhead. By contrast, Denali's virtual Ethernet requires only a single PIO to send or receive a packet.

To estimate the benefit of Denali's simple virtual Ethernet, we modified the guest OS device driver to perform as many PIOs as the pcnet32 driver. Doing so increased the packet reception cost by 18,381 cycles (10.9 ms) and the packet transmission cost by 22,955 cycles (13.7 ms). This increases the overhead of receiving a single 100-byte UDP packet by 42%.

4.2.4 Source Code Complexity

As a final measure of the impact of Denali's virtual architecture, we quantify the size of the Denali source tree relative to Linux. This comparison therefore gives an indication of how complex it is to implement an isolation kernel that exposes Denali's architecture, as compared to the complexity of implementing an OS that exports high-level abstractions. Code size is important because it serves as an indication of the size of the trusted computing base, and it also impacts how easily the system can be maintained, modified, and debugged over time.

We compared Denali against Linux 2.4.16. For fairness of comparison, we choose a subset of Linux files that comprise a "bare-bones" kernel: no module support, no SMP support, no power management, only the ext2 file system, limited driver support, and so on. We use semicolon count as the metric of source lines, to account for different coding conventions.

Denali contains 26,634 source lines, while Linux has 66,326 source lines (Figure 6a). Only a small fraction (18%) of the Denali source is consumed by the "core kernel"; the remainder is dedicated to device drivers and architecture-dependent routines. Although drivers are known to be more buggy than core kernel code [8], the drivers used by Denali and "bare-bones" Linux consist of mature source code that has not changed substantially over time, e.g., the IDE driver, terminal support, and PCI bus probing.

In Figure 6b, we present a breakdown of the core kernel sizes of Denali and Linux. The Linux core kernel is an order-of-magnitude larger than Denali. The majority of the difference is attributable to Linux's implementation of stable storage (the ext2 file system) and networking (TCP/IP) abstractions. By deferring the implementation of complex abstractions to guest operating systems, Denali realizes a substantial reduction in core kernel source tree size.

4.3 Denali at Scale

In this section, we characterize Denali's performance at scale. We first analyze two scaling bottlenecks, which we removed before performing application-level scaling experiments. We then analyze two applications with fairly different performance requirements and characteristics: a web server and the Quake II game server.

4.3.1 Scaling Bottlenecks

The number of virtual machines to which our isolation kernel can scale is limited by two factors: per-machine metadata maintained by the kernel when a VM has been completely paged out, and the working set size of active VMs.

Per-VM kernel metadata: To minimize the amount of metadata the isolation kernel must maintain for each paged-out VM, wherever possible we allocate kernel resources on demand, rather than statically on VM creation. For example, page tables and packet buffers are not allocated to inactive VMs. Table 2 breaks down the memory dedicated to each VM in the system. Each VM requires 8,472 bytes, of which 97% are dedicated to a kernel thread stack. Although we could use continuations [10] to bundle up the kernel stack after paging a VM out, per-VM kernel stacks have simplified our implementation. Given the growing size of physical memory, we feel this is an acceptable tradeoff: supporting 10,000 VMs requires 81 MB of kernel metadata, which is less than 4% of memory on a machine with 2GB of RAM.

VM working set size: The kernel cannot control the size of a VM's working set, and the kernel's paging mechanism may cause a VM to perform poorly if the VM scatters small memory objects across its pages. One instance where memory locality is especially important is the management of the mbuf packet buffer pool inside the BSD TCP/IP stack of our Ilwaco guest OS. Initially, mbufs are allocated from a large contiguous byte array; this "low entropy" initial state means that a request that touches a small number of mbufs would only touch a single page in memory. After many allocations and

Component	Size (bytes)
thread stack	8192
register file	24
swap region metadata	20
paging metadata	40
virtual Ethernet structure	80
pending alarms	8
VM boot command line	64
other	72
Total	**8472**

Table 2: **Per-VM kernel metadata:** this table describes the residual kernel footprint of each VM, assuming the VM has been swapped out.

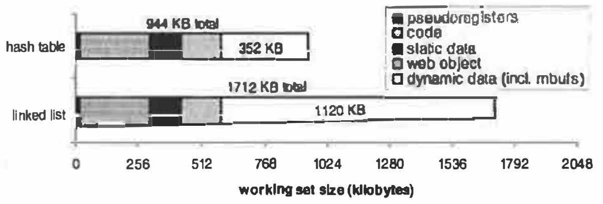


Figure 7: **Mbuf entropy and memory footprint:** eliminating mbuf entropy with a hash table can halve memory footprint.

deallocations from the mbuf pool, the default BSD implementation of the mbuf pool scatters back-to-back mbuf allocations across pages: in the worst case, as many pages are necessary as referenced mbufs, increasing the memory footprint of a VM.

We have observed the effects of mbuf entropy in practice, especially if a VM is subjected to a burst of high load. Figure 7 shows the effect of increasing the offered load on a web server inside a VM. The memory footprint of the VM using the default, linked list BSD implementation of the mbuf pool increases by 83% as the system reaches overload. We improved memory locality by replacing the linked list with a hash table that hashes mbufs to buckets based on the memory address of the mbufs; by allocating from hash buckets, the number of memory pages used is reduced. With this improvement, the VM's memory footprint remained constant across all offered loads. The savings in memory footprint resulted in nearly a factor of two performance improvement for large numbers of concurrent web server VMs.

More generally, the mbuf entropy problem is indicative of two larger issues inherent in the design of a scalable isolation kernel. First, the paging behavior of guest operating systems is a crucial component of overall performance; most existing OSs are pinned in memory and have little regard for memory locality. Second, memory allocation and deallocation routines (e.g., garbage collection) may need to be reexamined to promote memory locality; existing work

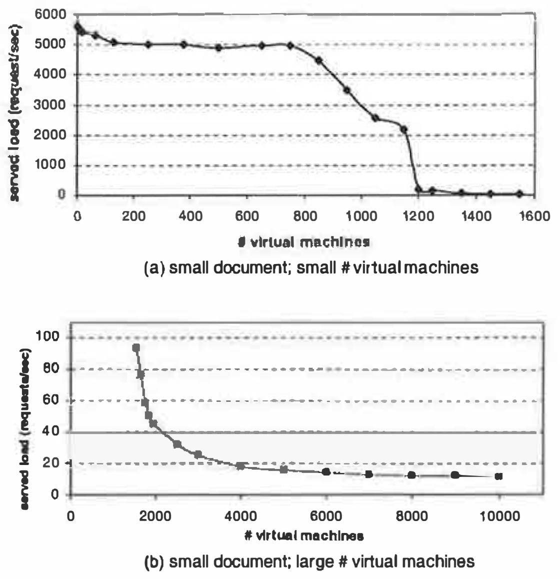


Figure 8: **In-core vs. out-of-core:** (a) shows aggregate performance up to the "cliff" at approximately 1000 VMs; (b) shows aggregate performance beyond the cliff.

on improving paging performance in object-oriented languages could prove useful.

4.3.2 Web server performance

To understand the factors that influence scalability for a throughput-centric workload, we analyzed Denali's performance when running many web server VMs. We found that three factors strongly influenced scalability: disk transfer block size, the popularity distribution of requests across VMs, and the object size transferred by each web server.

To evaluate these factors, we used a modified version of the httperf HTTP measurement tool to generate requests across a parameterizable number of VMs. We modified the tool to generate requests according to a Zipf distribution with parameter α. We present results for repeated requests to a small object of 2,258 bytes (approximately the median web object size). Requests of a larger web object (134,007 bytes) were qualitatively similar.

The performance of Denali at scale falls into two regimes. In the *in-core* regime, all VMs fit in memory, and the system can sustain nearly constant aggregate throughput independent of scale. When the number of active VMs grows to a point that their combined working sets exceed the main memory capacity, the system enters the *disk-bound* regime. Figure 8 demonstrates the sharp performance cliff separating these regimes.

In-core regime: To better understand the performance cliff, we evaluated the effect of two variables: disk block transfer size, and object popularity distribution. Reducing the block size used during

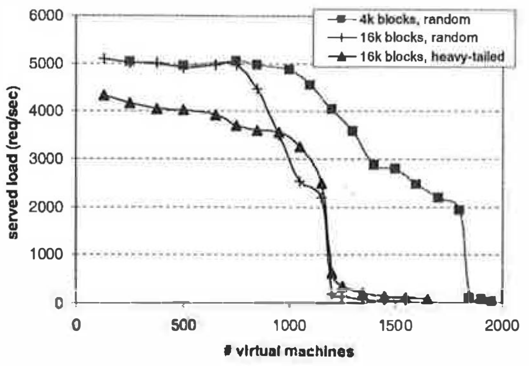


Figure 9: **Block size and popularity distribution:** this graph shows the effect of varying block size and popularity distribution on the "cliff"; the web servers were serving a 2,258 byte document.

paging can improve performance by reducing internal fragmentation, and as a consequence, reducing a VM's in-core footprint. This has the side-effect of delaying the onset of the performance cliff (Figure 9): by using a small block size, we can push the cliff to beyond 1000 VMs.

Disk-bound regime: To illustrate Denali's performance in the disk-bound regime, we examined web server throughput for 4,000 VMs serving the "small" document; the footprint of 4,000 VMs easily exceeds the size of main memory. Once again, we considered the impact of block size and object popularity on system performance.

To explore the effect of heavy-tailed distributions, we fixed the disk block transfer size at 32 kilobytes, and varied the Zipf popularity parameter α. As α increases, the distribution becomes more concentrated on the popular VMs. Unlike the CPU and the network, Denali's paging policy is purely demand driven; as a result, Denali is able to capitalize on the skewed distribution, as shown in Figure 10.

Figure 11 illustrates the effect of increased block size on throughput. As a point of comparison, we include results from a performance model that predicts how much performance our three disk subsystem should support, given microbenchmarks of its read and write throughput, assuming that each VM's working set is read in using random reads and written out using a single sequential write. Denali's performance for random requests tracks the modeled throughput, differing by less than 35% over the range of block sizes considered.

This suggests that Denali is utilizing most of the available raw disk bandwidth, given our choice of paging policy. For heavy-tailed requests, Denali is able to outperform the raw disk bandwidth by caching popular virtual machines in main memory. To improve performance beyond that which we have reported, the random disk reads induced by paging

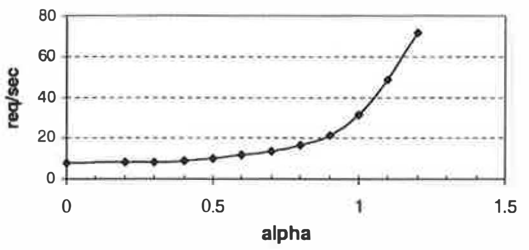


Figure 10: **Out-of-core performance vs. α:** increasingly skewed popularity distributions have better out-of-core performance; this data was gathered for 4,000 VMs serving the small web object, and a block size of 32KB.

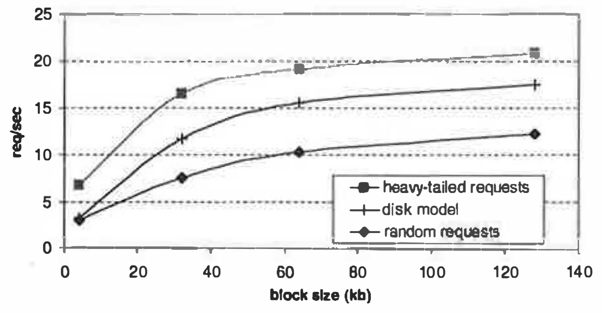


Figure 11: **Out-of-core performance vs. block size:** increased block size leads to increased performance in the out-of-core regime.

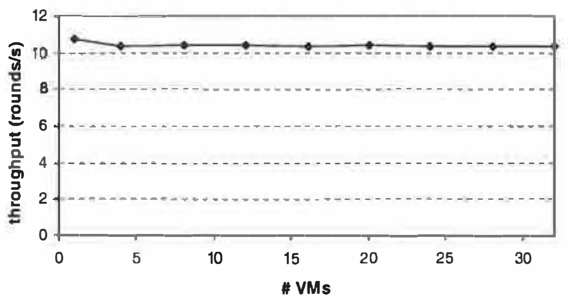
would need to be converted into sequential reads; this could be accomplished by reorganizing the swap disk layout so that the working sets of VMs are laid out sequentially, and swapped in rather than paged in.

4.3.3 Quake II server performance

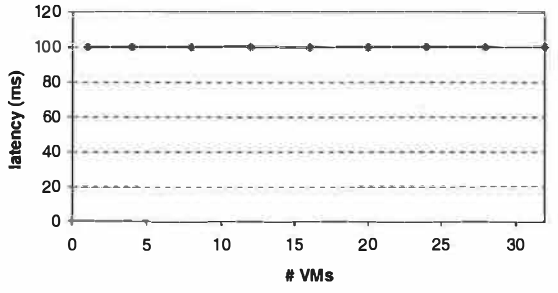
As a second workload to test the scalability of Denali, we ported the GPL'ed Quake II game server to Ilwaco. Quake II is a latency-sensitive multiplayer game. Each server maintains state for a single game session; clients participating in the session send and receive coordinate updates from the server. We use two metrics as a measure of the quality of the game experience: the *latency* between a client sending an update to the server and receiving back a causally dependent update, and the *throughput* of updates sent from the server. Steady latency and throughput are necessary for a smooth, lag-free game experience.

To generate load, we used modified Linux Quake II clients to play back a recorded game session to a server; for each server, we ran a session with four clients. As a test of scalability, we measured the throughput and latency of a Quake server as a function of the number of concurrently active Quake VMs. Figure 12 shows our results.

As we scaled up the number of VMs (and the number of clients generating load), the average throughput and latency of each server VM remained essentially constant. At 32 VMs, we ran out of client



(a) Quake II server throughput (per VM)



(b) Quake II server latency

Figure 12: **Quake II server scaling benchmarks:** even with 32 concurrent active Quake II server VMs, the throughput and latency to each server remained constant. At 32 servers, we ran out of client machines to drive load.

machines to generate load. Even with this degree of multiplexing, both throughput and latency remained constant, suggesting that the clients' game experiences would still be good.

Although the Quake server is latency-sensitive, it is in many ways an ideal application for Denali. The default server configuration self-imposes a delay of approximately 100 ms between update packets, in effect introducing a sizable "latency buffer" that masks queuing and scheduling effects in Denali. Additionally, because the server-side of Quake is much less computationally intensive than the client-side, multiplexing large numbers of servers is quite reasonable.

5 Related Work

We consider related work along two axes: operating system architectures, and techniques for isolating untrusted code.

5.1 OS architectures

The idea of hosting multiple isolated protection contexts on a single machine is not new: Rushby's separation kernel [28] is an instance of this idea. Denali puts these ideas into practice, and explores the systems issues when scaling to a large number of protection domains.

Exokernels [20] eliminate high-level abstractions to enable OS extensibility. Denali differs from Exokernels in its approach to naming: Denali exposes virtual, private name spaces, whereas Exokernels expose the physical names of disk, memory, and network resources. The Exokernel's global namespace allow resources to be shared freely, necessitating complex kernel mechanisms to regulate sharing.

Denali is similar to microkernel operating systems like Mach [1]. Indeed, Denali's VMs could be viewed as single-threaded applications on a low-level microkernel. However, the focus of microkernel research has been to push OS functionality into shared servers, which are themselves susceptible to the problems of high-level abstractions and data sharing. Denali emphasizes scaling to many untrusted applications, which was never an emphasis of microkernels.

Nemesis [23] shares our goal of isolating performance between competing applications. Nemesis adopts a similar approach, pushing most kernel functionality to user-level. Nemesis was not designed to sandbox untrusted code; Nemesis applications share a global file system and a single virtual address space.

The Fluke OS [13] proposes a recursive virtual machine model, in which a parent can re-implement OS functionality on behalf of its children. Like Denali, Fluke exposes private namespaces through its "state-encapsulation" property. The primary motivation for this is to support checkpointing and migration, though the security benefits are alluded to in [22]. Denali exposes a virtual hardware API, whereas Fluke virtualizes at the level of OS API. By virtualizing below abstractions, Denali's kernel is simple, and we avoid layer-below vulnerabilities.

Virtual machine monitors have served as the foundation of several "security kernels" [21]. More recently, the NetTop proposal aims to create secure virtual workstations running on VMWare [24]. Denali differs from these efforts in that we aim to provide scalability as well as isolation. We assume a weaker threat model; for example, we are not concerned with covert channels between VMs.

VMMs like Disco [6] and VM/370 [9] have the goal of supporting legacy systems, and therefore minimize architectural modifications to maintain compatibility. In comparison, isolation kernels rely on virtualization for isolation: backwards compatibility is not their primary goal. As a result, isolation kernels have the freedom to make strategic changes to the exposed virtual architecture for scalability, performance, and simplicity.

5.2 Enforcing isolation

Many projects provide OS support for isolating untrusted code, including system call interposition [16] and restricted execution contexts [32]. These proposals provide *mechanisms* for enforcing the principle of least privilege. However, expressing an appropriate access control *policy* requires a security expert to reason about access permissions to grant applications; this is a difficult task on modern systems with thousands of files and hundreds of devices. Denali imposes a simple security policy: complete isolation of VMs. This obviates the policy problem, and provides robust isolation for applications with few sharing requirements.

WindowBox [3] confines applications to a virtual desktop, imposing a private namespace for files. Because it is implemented inside a conventional OS, WindowBox's security is limited by high-level abstractions and global namespaces. For example, all applications have access to the Windows registry, which has been involved in many vulnerabilities.

Software VMs (like Java) have been touted as platforms for isolating untrusted code. Experience with these systems has demonstrated a tradeoff between security and flexibility. The Java sandbox was simple and reasonably secure, but lacked the flexibility to construct complex applications. Extensible security architectures [33] allow more flexibility, but reintroduce the problem of expressing an appropriate access control policy. Denali avoids this tradeoff by exposing a raw hardware API, complete with I/O devices, which allows VMs to build up arbitrary abstractions inside a guest OS. In addition, Denali's virtual architecture closely mirrors the underlying physical architecture, avoiding the need for a complex runtime engine or just-in-time compiler.

The problem of performance isolation has been addressed by server and multimedia systems [4, 26, 23]. Resource containers demonstrate that OS abstractions for resource management (processes and threads) are poorly suited to applications' needs. Denali's VMs provide a comparable resource management mechanism. We believe that isolation kernels can provide more robust performance isolation by operating beneath OS abstractions and data sharing. As an example, Reumann et al. conclude that there is no simple way to account for the resources in the file system buffer cache [26].

Finally, numerous commercial and open-source products provide support for virtual hosting, including freeVSD, Apache virtual hosts, the Solaris resource manager, and Ensim's ServerXchange. All work within a conventional OS or application, and therefore cannot provide the same degree of isolation as an isolation kernel. Commercial VMMs provide virtual hosting services, including VMWare's ESX server and IBM's z/VM system. By selectively modifying the underlying physical architecture, Denali can scale to many more machines for a given hardware base. We are not aware of detailed studies of the scalability of these systems.

6 Conclusions

This paper presented the design and implementation of the Denali isolation kernel, a virtualization layer that supports the secure multiplexing of a large number of untrusted Internet services on shared infrastructure. We have argued that isolation kernels are necessary to provide adequate isolation between untrusted services, and to support scaling to a large number of Internet services, as required by cost-efficiency. Quantitative evaluation of our isolation kernel has demonstrated that the performance overhead of virtualization is reasonable, that our design choices were both necessary and reasonable, and that our design and implementation can successfully scale to over 10,000 services on commodity hardware.

We believe that isolation kernels have the potential to dramatically change how Internet services are deployed. An isolation kernel allows a service to be "pushed" into third party infrastructure, thereby separating the management of physical infrastructure from the management of software services and lowering the barrier to deploying a new service.

7 Acknowledgments

We are grateful for the help of our shepherd, Robert Morris, whose suggestions have greatly improved the final draft of this paper. We would also like to thank Brian Bershad, David Wetherall, Neil Spring, Tom Anderson, John Zahorjan, and Hank Levy for their valuable feedback. This work also benefitted greatly from many discussions with the Systems and Networking group at Microsoft Research, particularly Bill Bolosky, Marvin Theimer, Mike Jones, and John Douceur. This work was supported in part by NSF Career award ANI-0132817, funding from Intel Corporation, and a gift from Nortel Networks.

References

[1] M. Accetta et al. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, 1986.

[2] Collaborative advanced interagency research network (cairn). `http://www.cairn.net`, 1997.

[3] D. Balfanz and D.R. Simon. Windowbox: A simple security model for the connected desktop. In *Proceedings of the 4th USENIX Windows Systems Symposium*, August 2000.

[4] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating system design and implementation*, February 1999.

[5] L. Breslau et al. Web caching, and Zipf-like distributions: Evidence, and implications, Mar 1999.

[6] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997.

[7] R. Carr and J. Hennessy. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proceedings of the 8th Symposium on Operating System Principles*, Dec 1981.

[8] A. Chou et al. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP '01)*, October 2001.

[9] R.J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5), 1981.

[10] R.P. Draves, B.N. Bershad, R.F. Rashid, and R.W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, USA, October 1991.

[11] P. Druschel and G. Banga. Lazy receiver processing: A network subsystem architecture for server systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, USA, Oct 1996.

[12] D. Ely, S. Savage, and D. Wetherall. Alpine: A user-level infrastructure for network protocol development. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems (USITS '01)*, March, 2001.

[13] B. Ford et al. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[14] B. Ford et al. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[15] A.F. Forin, D.B. Golub, and B.N. Bershad. An I/O system for Mach. In *Proceedings of the Usenix Mach Symposium (MACHNIX)*, Nov 1991.

[16] I. Goldberg, D. Wagner, R. Thomas, and E.A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the sixth USENIX Security Symposium*, July 1996.

[17] R.P. Goldberg. *Architectural Principles for Virtual Computer Systems.* PhD thesis, Harvard University, 1972.

[18] D. Gollmann. *Computer Security.* John Wiley and Son, Ltd., 1st edition, February 1999.

[19] Robert S. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. In *Proceedings of the Fourth Annual Usenix Tcl/Tk Workshop*, 1996.

[20] M.F. Kaashoek et al. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997.

[21] P.A. Karger et al. A retrospective on the VAX VMM security kernel. 17(11), November 1991.

[22] J. Lepreau, B. Ford, and M. Hibler. The persistant relevance of the local operating system to global applications. In *Proceedings of the Seventh SIGOPS European Workshop*, Sep 1996.

[23] I. Leslie et al. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7), 1996.

[24] R. Meushaw and D. Simard. NetTop: Commercial technology in high assurance applications. http://www.vmware.com/, 2000.

[25] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large-scale Internet measurement. *IEEE Communications Magazine*, 36(8):48–54, August 1998.

[26] J. Reumann et al. Virtual services: A new abstraction for server consolidation. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, USA, June 2000.

[27] J.S. Robin and C.E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

[28] J. Rushby. Design and verification of secure systems. In *Proceedings of the 8th Symposium on Operating System Principles*, December 1981.

[29] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1975.

[30] I.Stoica et al. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM 2001*, San Diego, USA, Aug 2001.

[31] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 Annual Usenix Technical Conference*, Boston, MA, USA, June 2001.

[32] M.M. Swift et al. Improving the granularity of access control in Windows NT. In *Proceedings of the 6th ACM Symposium On Access Control Models and Technologies*, May 2001.

[33] D.S. Wallach et al. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, Oct 1997.

[34] A. Wolman et al. Organization-based analysis of web-object sharing and caching. In *Proceedings of the 2nd USENIX Conference on Internet Technologies and Systems (USITS '99)*, Boulder, CO, Oct 1999.

ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay

George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, Peter M. Chen

Department of Electrical Engineering and Computer Science
University of Michigan
covirt@umich.edu, http://www.eecs.umich.edu/CoVirt

Abstract

Current system loggers have two problems: they depend on the integrity of the operating system being logged, and they do not save sufficient information to replay and analyze attacks that include any non-deterministic events. ReVirt removes the dependency on the target operating system by moving it into a virtual machine and logging below the virtual machine. This allows ReVirt to replay the system's execution before, during, and after an intruder compromises the system, even if the intruder replaces the target operating system. ReVirt logs enough information to replay a long-term execution of the virtual machine instruction-by-instruction. This enables it to provide arbitrarily detailed observations about what transpired on the system, even in the presence of non-deterministic attacks and executions. ReVirt adds reasonable time and space overhead. Overheads due to virtualization are imperceptible for interactive use and CPU-bound workloads, and 13-58% for kernel-intensive workloads. Logging adds 0-8% overhead, and logging traffic for our workloads can be stored on a single disk for several months.

1. Introduction

Improving the security of today's computer systems is an urgent and difficult problem. The complexity and rapid rate of change in current software systems prevents developers from verifying or auditing their code thoroughly enough to eliminate vulnerabilities. As a result, even the most diligent system administrators have to cope routinely with computer break-ins. This situation is likely to continue for the foreseeable future—statistics from the CERT® Coordination Center show a steady increase over the past 4 years in the number of incidents handled, the number of vulnerabilities reported, and the number of advisories posted [CER02].

The infeasibility of preventing computer compromises makes it vital to analyze attacks after they occur. Post-attack analysis is used to understand an attack, fix the vulnerability that allowed the compromise, and repair any damage caused by the intruder. Most computer systems try to enable this type of analysis by logging various events [Anderson80]. A typical Unix installation may record login attempts, mail processing events, TCP connection requests, file system mount requests, and commands issued by the superuser. Windows 2000 can record login/logoff events, file accesses, process start/exit events, security policy changes, and restart/shutdown events. Unfortunately, the audit logs provided by current systems fall short in two ways of what is needed: integrity and completeness.

Current system loggers lack *integrity* because they assume the operating system kernel is trustworthy; hence they are ineffective against attackers who compromise the operating system. One way current loggers trust the operating system is by keeping their logs on the local file system; this allows attackers who compromise the kernel to hide their activities by deleting past log records [CER01a]. Even if the existing log files are kept safely on another computer or on write-once media, attackers can forge misleading log records or prevent useful log records from being saved after they compromise the operating system. The absence of useful log records after the point of compromise makes it very difficult to assess and fix the damage incurred in the attack. It is ironic that current loggers work best when the kernel is not compromised, since audit logs are intended to be used when the system has been compromised!

Villains can attack kernels in many ways. The easiest way is to leverage the capabilities that the kernel provides to the superuser account. An attacker who has gained superuser privileges can change the kernel by writing to the physical memory through a special device (/dev/mem on Unix), by inserting a dynamically loaded kernel module, or by overwriting the boot sector or kernel image on disk. If an administrator has turned off these capabilities, an attacker can instead exploit a bug in the kernel itself. Kernels are large and complex and so tend to contain many bugs. In fact, a recent study used an automated tool to find over 100 security vulnerabilities in Linux and OpenBSD [Ashcraft02].

Current system loggers also lack *completeness* because they do not log sufficient information to recreate or understand all attacks. Typical loggers save only a few types of system events, and these events are often insufficient to determine with certainty how the break-in occurred or what damage was inflicted after the break-in. Instead, the administrator is left to guess what might have happened, and this is a painful and uncertain task. The attack analysis published by the Honeynet project typifies this uncertainty by containing numerous phrases such as "may indicate the method", "it seems reasonable to assume", "appears to", "likely edited", "presumably to", and "not clear what service was used" [Hon00].

More secure installations may log all inputs into the system, such as network activity or keyboard input. However, even such extensive logging does not enable an administrator to re-create attacks that involve *non-deterministic* effects. Many attacks exploit the unintended consequences of non-determinism (e.g. time-of-check to time-of-use race conditions [Bishop96])—recent advisories have described non-deterministic exploits in the Linux kernel, Microsoft Java VM, FreeBSD, NetBSD, kerberos, ssh, Tripwire, KDE, and Windows Media Services. Furthermore, the effects of non-deterministic events tend to propagate, so it becomes impossible to re-create or analyze a large class of events without replaying all earlier events deterministically. Encryption is a good example of this: encryption algorithms use non-deterministic events to generate entropy when choosing cryptographic keys, and all future communication depends on the value of the these keys. Without logging non-deterministic events, encrypted communication can be decrypted only if the attacker forgets to delete the key.

The goal of ReVirt is to solve the two problems with current audit logging. To improve the integrity of the logger, ReVirt encapsulates the target system (both operating system and applications) inside a virtual machine, then places the logging software beneath this virtual machine. Running the logger in a different domain than the target system protects the logger from a compromised application or operating system. ReVirt continues to log the actions of intruders even if they replace the target boot block or the target kernel.

To improve the completeness of the logger, ReVirt adapts techniques used in fault-tolerance for primary-backup recovery [Elnozahy02], such as checkpointing, logging, and roll-forward recovery. ReVirt is able to replay the complete, instruction-by-instruction execution of the virtual machine, even if that execution depends on non-deterministic events such as interrupts and user input. An administrator can use this type of replay to answer arbitrarily detailed questions about what transpired before, during, and after an attack.

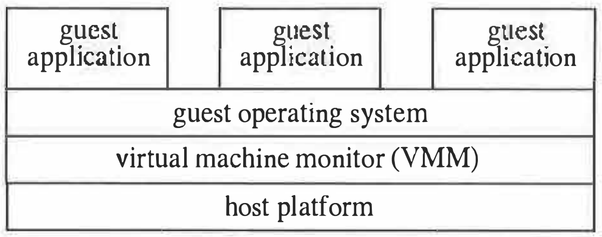


Figure 1: Virtual-machine structure.

2. Virtual machines

A virtual-machine monitor (VMM) is a layer of software that emulates faithfully the hardware of a complete computer system (Figure 1) [Goldberg74]. The abstraction created by the virtual machine monitor is called a virtual machine. The hardware emulated by the VMM is very similar (often identical) to the hardware on which the VMM is running, so the same operating systems and applications that run on the physical machine can run on the virtual machine. The host platform that the VMM runs on can be another operating system (the host operating system) or the bare hardware. The operating system running in the virtual machine is called the guest operating system to distinguish it from the host operating system running on the bare hardware. The applications running on top of the guest operating system are called guest applications to distinguish them from applications running on the host operating system (of which the VMM is one). The VMM runs in a separate domain from the guest operating system and applications; for example, the VMM may run in kernel mode and the guest software may run in user mode.

Our research group (CoVirt) is interested in enhancing security by running the target operating system and all target services inside a virtual machine (making them guest operating system and applications), then adding security services in the VMM or host platform [Chen01].

Of course, even the VMM may be subject to security breaches. Fortunately, the VMM makes a much better trusted computing base than the guest operating system, due to its narrow interface and small size. The interface provided by the VMM is identical or similar to the physical hardware (CPU, memory, disks, network card, monitor, keyboard, mouse), whereas the interface provided by a typical operating system is much richer (processes, virtual memory, files, sockets, GUIs). The narrow VMM interface restricts the actions of an attacker. In addition, the simpler abstractions provided by a VMM lead to a code size that is several orders of magnitude smaller than a typical operating system, and this smaller code size makes it easier to verify the VMM. As we will see, the narrow interface of the VMM also makes it easier to log and replay.

Virtual machines can be classified by how similar they are to the host hardware. At one extreme, traditional virtual machines such as IBM's VM/370 [Goldberg74] and VMware [Sugerman01] export an interface that is backward compatible with the host hardware (the interface is either identical or slightly extended). Operating systems and applications that were intended to run on the host platform can run on these VMMs without change. At the other extreme, language-level virtual machines like the Java VM export an interface that is completely different from the host hardware. These VMMs can run only operating systems and applications written specifically for them.

Other virtual machines such as the VAX VMM security kernel [Karger91] fall somewhere in the middle—they export an interface that is similar but not identical to the host hardware [Bellino73]. These types of VMMs typically deviate from the host hardware interface when interacting with peripherals. Virtualizing the register interface to peripherals controllers is difficult and time consuming, so many virtual machines provide higher-level methods to invoke I/O. A guest operating system must be ported to run on these VMMs. Specifically, the device drivers in the guest kernel must use the higher-level methods in the VMM; e.g. a disk device driver might use the host system calls `read` and `write` to access the virtual hard disk. The work required to port a guest operating system to these types of VMMs is similar to that done by device manufacturers who write drivers for their devices.

3. UMLinux

ReVirt uses a virtual machine called UMLinux [Buchacker01].[1] UMLinux falls in the last category of virtual machines; the VMM in UMLinux exports an interface that is similar but not identical to the host hardware. The version of UMLinux described and used in

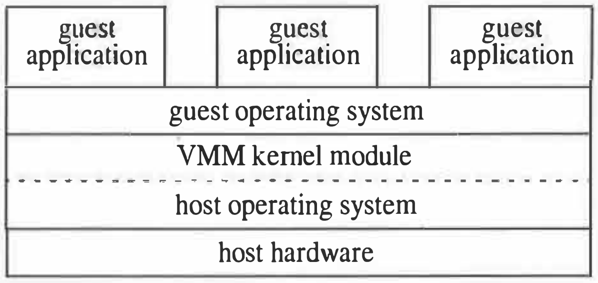


Figure 2: UMLinux OS-on-OS structure. Our version of UMLinux is implemented as a loadable kernel module in the host operating system. The device and interrupt drivers in the guest operating system use host services such as system calls and signals.

this paper is modified from code developed by researchers at the University of Erlangen-Nürnberg. Our version of the UMLinux VMM uses custom optimizations in the underlying operating system to achieve an order of magnitude speedup over the original UMLinux [King02].

3.1. UMLinux structure and operation

The virtual machine in UMLinux runs as a user process on the host. Both the guest operating system and all guest applications run inside this single host process (the virtual-machine process). The guest operating system in UMLinux runs on top of the host operating system and uses host services (e.g. system calls and signals) as the interface to peripheral devices (Figure 2). We call this virtualization strategy *OS-on-OS*, and we call the normal structure where target applications run directly on the host operating system *direct-on-host*. The guest operating system used in this paper is Linux 2.4.18, and the host operating system is also Linux 2.4.18.[2]

The VMM in our version of UMLinux is implemented as a loadable module in the host Linux kernel, plus some hooks in the kernel that invoke our VMM module. The VMM module is called before and after each signal and system call to/from the virtual-machine process.

Most instructions executed within the virtual machine execute directly on the host CPU. Memory accesses are translated by the host's MMU based on

1. Note that UMLinux is different from the similarly-named User-Mode Linux [Dike00].

2. The guest and host operating systems can also be different. We use the same operating system for guest and host to enable a more direct comparison between running applications on the UMLinux guest and running applications directly on the host.

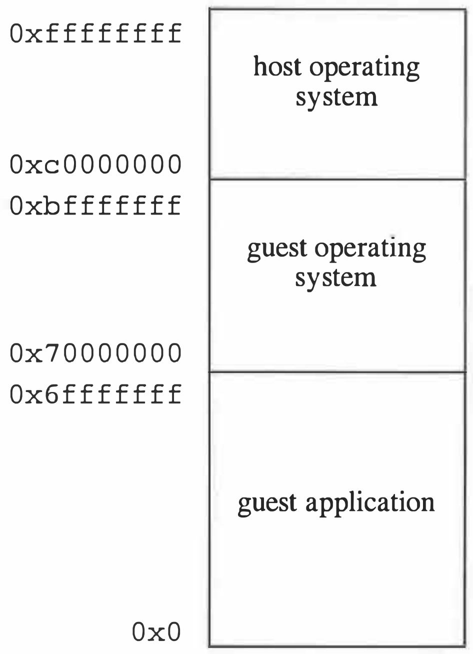


Figure 3: UMLinux address space. As with all Linux processes, the host kernel address space occupies [0xc0000000, 0xffffffff], and the host user address space occupies [0x0, 0xc0000000). The guest kernel occupies the upper portion of the host user space [0x70000000, 0xc0000000), and the current guest application occupies the remainder of the host user space [0x0, 0x70000000).

Host component or event	Emulation mechanism in UMLinux
hard disk	host raw partition
CD-ROM	host /dev/cdrom
floppy disk	host /dev/floppy
network card	TUN/TAP virtual Ethernet device
console	TCP to host application
video card	none (display to remote X server)
current privilege level	VMM variable
system calls	SIGUSR1 signal
timer interrupts	timer + SIGALRM signal
I/O device interrupts	SIGIO signal
memory exception	SEGV signal
enable/disable interrupts	mask signals

Table 1: Mapping between host components and UMLinux equivalents.

translations that are set up via the host operating system's `mmap`, `munmap`, and `mprotect` system calls.

Figure 3 shows the address space of the virtual-machine process. Host memory protections are used to prevent guest applications from accessing the guest kernel's address space.

UMLinux provides a software analog to each peripheral device in a normal computer system. Table 1 shows the mapping from each host component or event to its software analog in the virtual machine. UMLinux uses a host file or raw device to emulate the hard disk, CD-ROM, and floppy. Our version of UMLinux uses the TUN/TAP virtual Ethernet device in Linux to emulate the network card. UMLinux uses a small X application on the host to display console output and read keyboard input; this application communicates with the guest kernel's console driver via TCP. UMLinux uses no video card; instead it displays graphical output to a remote X server (which would typically be the host's X server).

UMLinux provides a software analog to the computer's current privilege level. The VMM module maintains a virtual privilege level, which is set to *kernel* when transferring control to the guest kernel, and is set to *user* when transferring control to a guest application. The VMM module uses the current virtual privilege level to distinguish between system calls issued by a guest application and system calls issued by the guest kernel.

System calls issued by a guest application must be redirected to the guest kernel's system-call trap handler. When a guest application executes a system-call instruction (`int 0x80`), the host CPU traps to the host kernel's system-call handler, which then transfers control to the VMM kernel module. If the current virtual privilege level is set to *kernel*, then the VMM knows the guest kernel made the system call (typically to access a host device or change memory translations). In this case, the VMM checks that this system call is one that a UMLinux guest kernel is expected to make, then passes it through to the host kernel. If the virtual privilege level is set to *user*, then the VMM knows a guest application made the system call. In this case, the VMM module notifies the guest kernel by sending it a signal (SIGUSR1). The VMM module passes the registers at the time of the trap to the guest kernel's signal handler. The SIGUSR1 signal handler in the guest kernel is the equivalent of the system-call trap handler in a normal operating system.

SIGALRM, SIGIO, and SIGSEGV signals are used to emulate the hardware timer, I/O device interrupts, and memory exceptions. As with SIGUSR1, the host kernel delivers these signals to the registered signal handler in the guest kernel. These signal handlers are the equivalent of the timer-interrupt, I/O-interrupt, and memory exception handlers in a normal operating system.

UMLinux emulates the enabling and disabling of interrupts by masking signals (using the `sigprocmask` system call).

3.2. Trusted computing base for UMLinux

All the virtualization strategies described in Section 2 depend on the trustworthiness of all layers below the guest operating system (the VMM and host platform in Figure 1). For UMLinux, the trusted computing base (TCB) is comprised of the VMM kernel module and the host operating system. UMLinux's TCB is larger than the TCB for virtual machines that run directly on the hardware, such as IBM's VM/370 or VMware's ESX Server. UMLinux's TCB is similar to other virtual machines that cooperate with a host operating system, such as VMware Workstation.

A common question is whether a security service that is added to the host operating system in an OS-on-OS structure is more protected from attack than a security service that is added to the host operating system in a direct-on-host structure. For example, while the logging in an OS-on-OS structure does not depend on the integrity of the guest operating system, doesn't it still depend on the integrity of the host operating system?

We contend that the logging in an OS-on-OS structure is much more difficult to attack than the logging in a direct-on-host structure, because the TCB for an OS-on-OS structure can be much smaller than the complete host operating system [Meushaw00]. While both OS-on-OS and direct-on-host depend on the host operating system, the avenues a villain can use to attack the host differ greatly between the two structures.

Assume for this comparison that the villain has gained control over all target applications and can send arbitrary network packets to the host. A villain can launch attacks against the host operating system from two directions. First, a villain can attack from above by causing application processes to invoke the host operating system in dangerous ways. In a direct-on-host structure, the attacker has complete freedom to invoke whatever functionality the host operating system makes available to user processes. The attacker can control multiple application processes, access multiple files, and issue arbitrary system calls. In an OS-on-OS structure, an attacker who has gained control of all application processes can use these same avenues to attack the *guest* operating system. However, even if the attacker gains control over the guest operating system, he/she is still severely restricted in the actions he/she can take against the *host* operating system. The guest kernel needs only a small subset of the functionality available to general-purpose host processes, and the VMM can easily disallow functionality outside this subset [Goldberg96]. For example, an attacker who has gained control over all target applications and the guest operating system still controls only a single host process (the virtual-machine process), can access only a few host files/devices (the virtual hard disk, the virtual CD-ROM, and the virtual floppy), and can make only a few system calls.

Second, a villain can attack the low level of the network protocol stack by sending dangerous network packets to the host (e.g. ping-of-death). As with attacks from above, less of the host operating system is exposed to dangerous packets with an OS-on-OS structure than a direct-on-host structure. Without virtual machines, packets traverse through the entire network stack and are delivered to applications; villains can thus craft packets to attack any layer of the network stack. With virtual machines, packets need only traverse a small part of the network stack.

The portion of the host operating system included in UMLinux's TCB is the host OS code that the guest kernel or incoming packets can invoke (plus the VMM, which disallows invocations outside this portion). We have yet to measure the size of this code rigorously, but early indications suggest that this portion is significantly smaller than the entire host operating system. For example, our VMM restricts the guest kernel to use fewer than 7% of the system calls available to general host processes, and network traffic to the virtual machine is processed mostly by the guest operating system's TCP and UDP stacks (only a small IP-layer packet filter is used in the host operating system).

The TCB of our current UMLinux prototype, while smaller than the complete host operating system, is not yet as small as it could be. The host operating system in our prototype runs other processes which could be attacked (e.g. the X server), and network messages to these host processes traverse the entire host network stack. Our future work includes measuring and reducing the size of the host operating system used to support UMLinux. For example, we could further restrict the system calls issued by the guest kernel to use only certain parameter values, and we could move the X server into another virtual machine.

4. Logging and replaying UMLinux

4.1. Overview

Logging is used widely for recovering state. The basic concept is straightforward: start from a checkpoint of a prior state, then roll forward using the log to reach the desired state. The type of system being recovered determines the type of information that needs to be logged: database logs contain transaction records, file system logs contain file system data. Replaying a process requires logging the non-deterministic events that affect the process's computation. These log records guide the process as it re-executes (rolls forward) from a checkpoint. Most events are deterministic (e.g. arithmetic, memory, branch instructions) and do not need to be logged; the process will re-execute these events in the same way during replay as it did during logging.

Non-deterministic events fall into two categories: time and external input. Time refers to the exact point in the execution stream at which an event takes place. For example, to replay an interrupt, we must log the instruction at which the interrupt occurred. External input refers to data received from a non-logged entity, such as a human user or another computer. External input enters the processor via a peripheral device, such as a keyboard, mouse, or network card.

Note that output to peripherals does not affect the replaying process and hence need not be saved (in fact, output to peripherals will be reconstructed during replay). Non-determinism in the micro-architectural state (e.g. cache misses, speculative execution) also need not be saved, unless it affects the architectural state. Replaying a shared-memory multiprocessor requires saving the fine-grained interleaving order of memory operations and is outside the scope of this paper [LeBlanc87].

4.2. ReVirt

This section describes how we apply the general concepts of logging to enable replay of UMLinux running on x86 processors. ReVirt is implemented as a set of modifications to the host kernel.

Before starting UMLinux, we checkpoint the state by making a copy of its virtual disk. We currently require replay to start from a powered-off virtual machine, so the virtual disk comprises all state in the virtual machine. We envision checkpointing being a rare event (once every few days), so copying speed is not critical.

Log records are added and saved to disk in a manner similar to that used by the Linux `syslogd` daemon. The VMM kernel module and kernel hooks add log records to a circular buffer in host kernel memory, and a user-level daemon (`rlogd`) consumes the buffer and writes the data to a log file on the host.

ReVirt must log all non-deterministic events that can affect the execution of the virtual-machine process. Note that many non-deterministic host events do not need to be logged, because they do not affect the execution of the virtual machine. For example, host hardware interrupts do not affect the virtual-machine process unless they cause the host kernel to deliver a signal to the virtual-machine process. Likewise, the scheduling order of other host processes does not affect the virtual-machine process because there is no interprocess communication between the virtual-machine process and other host processes (no shared files, memory, or messages).

ReVirt does have to log asynchronous virtual interrupts (synchronous exceptions like SIGSEGV are deterministic and do not need to be logged). Before delivering a SIGALRM or SIGIO host signal (representing virtual timer and I/O interrupts) to the virtual-machine process, ReVirt logs sufficient information to re-deliver the signal at the same point during replay. To uniquely identify the interrupted instruction, ReVirt logs the program counter and the number of branches executed since the last interrupt [Bressoud96]. Because the x86 architecture allows a block memory instruction (repeat string) to be interrupted in the middle of its execution, we also must log the register (`ecx`) that stores the number of iterations remaining at the time of the interrupt.

x86 processors provide a hardware performance counter that can be configured to compute the number of branches that have executed since the last interrupt [Int01]. The `branch_retired` configuration of this performance counter on the AMD Athlon processor counts branches, hardware interrupts (e.g. timer and network interrupts), faults (e.g. page faults, memory protection faults, FPU faults), and traps (e.g. system calls). We use another hardware performance counter to count the number of hardware interrupts and subtract this from the `branch_retired` counter. Similarly, we instrument the host kernel to count the number of faults and traps and subtract this from the branch_retired counter. We configure the branch_retired counter to count only user-level branches. This makes it easier to count the number of branches precisely, because it keeps the count independent of the code executed in the kernel interrupt handlers.

In addition to logging asynchronous virtual interrupts, ReVirt must also log all input from external entities. These include most virtual devices: keyboard, mouse, network interface card, real-time clock, CD-ROM, and floppy. Note that input from the virtual hard disk is deterministic, because the data on the virtual hard disk will be reconstructed and re-read during replay. One can imagine requiring the user to re-insert the same floppy disk or CD-ROM during replay, in which case reads from the CD-ROM and floppy would also be deterministic and would not need to be logged. However, we do not expect data from these sources to be a significant portion of the log, because these data sources are limited in speed by the user's ability to switch media.[3]

The UMLinux guest kernel reads these types of input data by issuing host system calls recv, read, and gettimeofday. The VMM kernel module logs the input data by intercepting these system calls. In general, ReVirt must log any host system call that can yield non-deterministic results.

The x86 architecture includes a few instructions that can return non-deterministic results, but that do not normally trap when running in user mode. Specifically, the x86 rdtsc (read timestamp counter) and rdpmc (read performance monitoring counter) instructions are difficult for us to log. To make the virtual-machine process completely deterministic during replay, we set the processor control register (CR4) to trap when these instructions are executed. We remove the guest kernel's rdtsc instructions by replacing them with a gettimeofday host system call (and scaling the result); it would also be possible to leave these calls in the guest kernel, then trap, emulate, and log the rdtsc instruction. We disallow rdpmc in the guest kernel and guest applications.

During replay, ReVirt prevents new asynchronous virtual interrupts from perturbing the replaying virtual-machine process. ReVirt plays back the original asynchronous virtual interrupts using the same combination of hardware counters and host kernel hooks that were used during logging. ReVirt goes through two phases to find the right instruction at which to deliver the original asynchronous virtual interrupt. In the first phase, ReVirt configures the branch_retired performance counter to generate an interrupt after most (all but 128) of the branches in that scheduling interval. In the second phase, ReVirt uses breakpoints to stop each time it executes the target instruction. At each breakpoint, ReVirt compares the current number of branches with the desired amount. The first phase executes at the same speed as the original run and is thus faster than the second phase, which triggers a breakpoint each time the target instruction is executed. The second phase is needed to stop at exactly the right instruction, because the interrupt generated by the branch_retired counter does not stop execution instantaneously and may execute past the target number of branches.

Replay can be conducted on any host with the same processor type as the original host. Replaying on a different host allows an administrator to minimize downtime for the original host.

4.3. Cooperative logging

Most sources of non-determinism generate only a small amount of log data. Keyboard and mouse input is limited by the speed of human data entry. Interrupts are relatively frequent, but each interrupt generates only a few bytes of log data. Of all the sources of non-determinism, only received network messages have the potential to generate enormous quantities of log data.

We can reduce the amount of logged network data with a simple observation: one computer's received message is another computer's sent message. If the sending computer is being logged via ReVirt, then the receiver need not log the message data because the sender can re-create the sent data via replay. This technique is used commonly in message-logging recovery protocols [Elnozahy02] and can be viewed as expanding the domain of the replay system to include other computers. Thus the receiver need not log data sent from computers that can cooperate in the replay; the receiver need only log a unique identifier for the message (e.g. the identity of the sending computer and a sequence number).

Cooperative logging can reduce the amount of logged network data dramatically in certain cases. For example, if all computers on a LAN participate, then only traffic from outside the LAN needs to be logged, thus reducing the maximum log growth rate from LAN bandwidths to WAN bandwidths.

3. If the CD-ROM is switched by an automated jukebox, then the jukebox can participate in replay and CD-ROM reads can be considered deterministic.

While cooperating logging can reduce log volume, it complicates replay and requires that cooperating computers trust each other to regenerate the same message data during replay. We have not yet implemented cooperative logging in ReVirt.

4.4. Alternative architectures for logging and replay

We considered several strategies for building a logging/replay system before settling on the virtual-machine approach described above. In particular, we started by implementing a direct-on-host system, where the host kernel logged and replayed all its host processes. As discussed in Section 3.2, the direct-on-host approach is not as secure as a virtual-machine approach. We also found it to be much more difficult to log and replay all host processes than to log and replay a virtual-machine process. Interestingly, the narrow interface (between UMLinux and the host kernel) that makes an OS-on-OS approach more secure than a direct-on-host structure also makes an OS-on-OS system easier to replay.

The general approach for replaying a direct-on-host system is similar to that used in ReVirt: the system must log and replay all non-determinism. The same types of non-determinism exist for multiple host processes as for our virtual-machine approach (interrupt timing, external input).

However, it is much more challenging to log and replay a direct-on-host structure than a virtual-machine process, because a direct-on-host structure involves multiple host processes while an OS-on-OS approach involves only a single host process. (While the scheduling order between guest processes in UMLinux is non-deterministic, this is an abstraction above the virtual machine and is replayed deterministically as a result of deterministic signal delivery to the virtual-machine process.)

Replaying multiple host processes can be done in two ways, both of which are problematic. First, one can replay the communication channels between processes, but replaying a shared-memory communication channel requires complex instrumentation of the executing code and adds significant overhead [Netzer94].

Second, one can replay the scheduling order between host processes [Russinovich96]. This strategy is difficult because a host process can be interrupted while executing in kernel mode (e.g. while executing a system call). It is hard to identify the point in the kernel where an interrupt occurred, yet identifying this point is critical for replaying the exact scheduling order. The hardware performance counters we used to identify the exact interrupt point in ReVirt do not work well when the interrupt point is in the kernel, because we configure them to count only user-mode instructions. Configuring them to count both user and kernel instructions also leads to difficulties—the kernel does not execute deterministically, so the instruction counts would differ during replay.

A few solutions are possible, though none is appealing. First, one could change the host kernel to only allow interrupts at a few well-defined points and log which of these points was interrupted. This would require widespread changes to the host kernel. Second, one could try to replay the entire host kernel. This would require changing the interrupt handlers to log and replay hardware interrupts, and adjusting the hardware performance counters for the different code paths executed by the interrupt handlers during logging and replay.

In addition to coping with scheduling order between multiple host processes, a direct-on-host approach must cope with a large number of non-deterministic interfaces. There are a large number of system calls, including some (e.g. `ioctl`) that have a very wide variety of possible parameters. Replaying a direct-on-host system requires one to identify, log, and replay non-determinism in each of these system calls. In contrast, ReVirt only needs to handle the few systems calls used by UMLinux.

4.5. Using ReVirt to analyze attacks

ReVirt enables an administrator to replay the complete execution of a computer before, during, and after the attack. Two types of tools can be built on this foundation to assist the administrator to understand the attack.

The first type of tool runs inside the guest virtual machine. ReVirt supports the ability to continue live (i.e. non-replaying) execution at any point in the replay. An administrator can use this ability to run new guest commands to probe the virtual machine state. For example, the administrator can stop the replay after a suspicious point and use normal guest commands to edit the current files, list the current processes, and debug processes. However, the virtual machine cannot switch back to replaying after being perturbed in this manner, because the instruction counts will not apply to the revised state. To continue the replay beyond the perturbed point, the analyst should checkpoint the process before perturbing it or start the replay over and let it continue to the later point.

Second, tools such as debuggers and disk analyzers can run outside the guest virtual machine and analyze the state of a virtual machine (address space, registers, and disk data). The advantage of these off-line tools is that they do not depend on the guest kernel or guest applications. For example, an off-line tool can inspect the contents of a directory even if the attacker has replaced the command that normally lists the directory.

A particularly useful tool that runs outside the guest is one that re-displays the original graphical output. Recall that UMLinux uses a remote X server (perhaps running on the host) as its graphical display. The replaying virtual-machine process faithfully recreates the stream of network packets being sent to the X server. However, the X server is not under the control of the replay system and will likely send back different packets to the virtual machine (e.g. due to different mouse movements). The packets being sent to the virtual machine do not affect replay, because the replaying machine gets its input packets from the log. However, the TCP protocol at the X server may expect different replies to the packets it sends to the virtual machine and may be confused by the virtual machine's resent packets. We address this with a simple X proxy on the host that opens a *new* TCP connection to the X server. The X proxy's goal is to act as a new X client that happens to send the same display messages to the X server as the virtual machine did during logging. The X proxy accomplishes this by receiving the packets being (re)sent from the replaying virtual machine, stripping off the Ethernet, IP, and TCP headers from these packets, reconstituting the X window data stream, and sending the data stream to the X server. Fortunately, the X protocol is largely deterministic and does not require the client to reply to messages sent from the X server (the sole exception is the X authentication protocol, and the X proxy can be written to navigate through this protocol).

5. Experiments

This section validates correctness and quantifies virtualization and logging overhead for our modified UMLinux and the ReVirt logging and replay system. All experiments are run on a computer with a AMD Athlon 1800+ processor, 256 MB of memory, and a Samsung SV4084 IDE disk. The guest kernel is Linux 2.4.18 ported to UMLinux, and the host kernel for UMLinux is a modified version of Linux 2.4.18. The virtual machine is configured to use 192 MB of "physical" memory. The virtual hard disk is stored on a raw disk partition on the host to avoid double buffering the virtual disk data in the guest and host file caches, and to prevent the virtual machine from benefitting unfairly from the host's file cache.

We evaluate our system on five workloads. All workloads start with a warm guest file cache. *POV-Ray* is a CPU-intensive ray-tracing program. We render the benchmark image from the POV-Ray distribution at quality 8. *kernel-build* compiles the complete Linux 2.4.18 kernel (make clean, make dep, make bzImage). *NFS kernel-build* is the same as *kernel-build* with the kernel stored on an NFS server. *SPECweb99* is a benchmark that measures web server performance; we use the 2.0.36 Apache web server. We configured *SPECweb99* with 15 simultaneous connections spread over two clients connected to a 100 Mb/s Ethernet switch. Both workloads exercise the virtual machine intensively by making many system calls. They are similar to the I/O-intensive and kernel-intensive workloads used to evaluate Cellular Disco [Govil00]. We also used ReVirt and UMLinux as the first author's desktop machine for 24 hours to get an idea of the virtualization and logging overhead for day-to-day use.

Each result represents the average of three runs (except for the daily-use test, which represents a single 24-hour period). Variance across runs is less than 3%.

5.1. Virtualization overhead

Our first concern is the time overhead that arises from running all applications in the UMLinux virtual machine. We compare running all applications within UMLinux with running them directly on a host Linux 2.4.18 kernel. The host and guest file systems have the same versions of all software exercised in the tests (based on RedHat 6.2).

Table 2 shows the results. UMLinux with our host optimizations adds very little overhead for compute-intensive applications such as *POV-Ray*. We also perceive no overhead when using UMLinux for interactive jobs such as e-mail, editing, word processing, and web browsing.

The overheads for *SPECweb99*, *kernel-build,* and *NFS kernel-build* are higher because they issue more guest kernel calls, each of which must be trapped by the VMM kernel module and reflected back to the guest kernel by sending a signal. In addition, *kernel-build* and *NFS kernel-build* cause a large number of guest processes to be created, each of which maps its executable pages on demand. Each demand-mapped page causes a signal to be delivered to the guest kernel, which must then ask the host kernel to map the new page.

Workload	UMLinux runtime (normalized to direct-on-host)
POV-Ray	1.01
kernel-build	1.58
NFS kernel-build	1.44
SPECweb99	1.13
daily use	≈ 1

Table 2: Virtualization overhead. This table shows the overhead caused by running applications in UMLinux. Runtime is normalized to the runtime when running directly on the host.

We believe the overheads for using UMLinux are low enough to be unnoticeable for normal desktop use. While overheads are higher for workloads that use the guest kernel intensively, we believe that even an overhead of 58% is not prohibitive for sites that value security.

For reference, VMware Workstation 3.1 has a normalized runtime of approximately 1.25 for *kernel-build*, UMLinux without our modifications to the host kernel has a normalized running time of 26, and a recent version of User-Mode Linux (configured to protect the guest kernel memory from guest applications) has a normalized runtime of 14. The low overhead of VMware and its acceptance in production settings indicate that virtualization can be made fast enough to enable services such as ReVirt.

5.2. Validating ReVirt correctness

Our next goal was to verify that the ReVirt system faithfully replays the exact execution of the original run. For these runs, we add extensive error checking to alert us if the replaying run deviated from the original. At every system call and virtual interrupt, we log all register values and the `branch_retired` counter and verify that these values are the same during replay. In addition, ReVirt's mechanism for replaying interrupts verifies that the branch count at the interrupted instruction matches the branch count seen at that instruction during logging.

We first run two micro-benchmarks in the virtual machine to verify that virtual interrupts are being replayed at the same point at which they occurred during logging. The first micro-benchmark runs two guest processes that share an mmap'ed memory region. Each guest process increments a shared variable 10,000,000 times, prints the resulting value, then repeats. Because the two guest processes share this variable, the output of process A depends on how many iterations process B executed by the time process A prints the value. The second microbenchmark runs a single process that increments a variable in an infinite loop. The process prints the current value when it receives a signal. This test verifies that the guest kernel re-delivers the signal at the same point as during logging.

We ran each micro-benchmarks 5 times, and each time the output during replay matched the original output, and all error checks passed.

We next run a macro-benchmark to verify that ReVirt faithfully plays back input from external systems and to exercise the system as a whole for longer periods. During the macro-benchmark, we boot the computer, start the GNOME window manager (displaying on a remote X server), open several interactive terminal windows, and concurrently build two applications (`freeciv` and `mup`) on a remote NFS server. The logging run of this benchmark generates 15,000,000 system calls and 55,000 virtual interrupts. ReVirt replayed this run without any deviation from the original run.

For the other tests used in this paper, we disable the extra error checking mentioned above. However, ReVirt always checks that the branch count at the interrupted instruction matches the branch count seen at that instruction during logging, and we have found this to detect errors effectively while we were developing ReVirt.

5.3. Logging and replaying overhead

Next we seek to quantify the space and time overhead of logging. We do not include the time and space overhead to checkpoint the system, since we expect a checkpoint to be amortized over a long period of time (e.g. a few days). Table 3 shows the time and space overhead for logging on the *POV-Ray, kernel-build, NFS*

Workload	Runtime with logging (normalized to UMLinux *without* logging)	Log growth rate	Replay runtime (normalized to UMLinux *with* logging)
POV-Ray	1.00	0.04 GB/day	1.01
kernel-build	1.08	0.08 GB/day	1.02
NFS kernel-build	1.07	1.2 GB/day	1.03
SPECweb99	1.04	1.4 GB/day	0.88
daily use	≈ 1	0.2 GB/day	0.03

Table 3: Time and space overhead of logging and replay. Logging slowdown shows the overhead caused by logging, relative to running UMLinux without logging. Log growth rate shows the average rate of growth of the log during the workload. Replay runtime is normalized to the runtime of UMLinux with logging. Replay runtime values less than 1 indicate that replay ran faster than logging, due to replay's ability to skip over periods of idle time.

kernel-build, and *SPECweb99* workloads. Logs are stored in a compressed format using `gzip`.

Table 3 shows that the time overhead of logging is small (at most 8%).

The space overhead of logging is small enough to save the logs over a long period of time at low cost. Workloads with little non-determinism (e.g. *POV-Ray*, *kernel-build*) generate very little log traffic. Note that the log data needed to replay local compilations takes much less space than the disk data generated in compilation.

The log growth rate for *NFS kernel-build* and *SPECweb99* is higher because of the need to log incoming network packets. However, it is still not prohibitive. For example, a 120 GB disk can store the volume of log traffic generated by *NFS kernel-build* for 3-4 months. If the file server used ReVirt, using cooperative logging at the client would reduce the log volume generated by *NFS kernel-build* to that of *kernel-build*.

We also used ReVirt and UMLinux as the first author's desktop machine for 24 hours to get an idea of the virtualization and logging overhead for day-to-day use[4]. We experienced no perceptible time overhead relative to running directly on the host, and the log occupied 0.2 GB after one day.

Table 3 shows that workloads typically replay at the same speed as they ran during logging. It is possible to replay a workload faster (sometimes much faster) than it ran during logging because replay skips over periods of idle time, such as that encountered during the non-working hours of the daily use workload.

4. This test was run using Linux 2.2.20 as the guest operating system.

5.4. Analyzing an attack

Finally, we demonstrate the ability of ReVirt to help analyze a non-deterministic attack that involves a kernel-level vulnerability. We re-introduced into our guest kernel the ptrace race condition that was present in Linux kernels before 2.2.19 [CER01b]. A villain exploits this bug by running a setuid process and attaching to it via `ptrace`. The vulnerability is non-deterministic because it depends on a time-of-check to time-of-use race condition. The attack is successful only if the file is not currently in the file cache, and the file cache state depends on the scheduling order and behavior of prior processes.

We exercised the vulnerability until compromising the system, then we added a trojan horse to /bin/ls and a backdoor to /etc/inetd.conf. ReVirt successfully replays the attack and allows us to find out how the attacker compromised the system and assess all damage done after the point of compromise. We were able to stop the replay after each point in the attack, run guest programs that examined the system state, and diagnose the method and effects of the intrusion.

6. Related work

Bressoud and Schneider's work on hypervisor-based fault tolerance [Bressoud96] shares several techniques with ReVirt. Bressoud and Schneider use a virtual machine for the PA-RISC architecture to interpose a software layer between the hardware and an unchanged operating system, and they log non-determinism to reconstruct state changes from a primary computer onto its backup.

While ReVirt shares several mechanisms with Hypervisor, ReVirt uses them to achieve a different and new goal. Hypervisor is intended to help tolerate faults by mirroring the state of a primary computer onto a backup. ReVirt takes some of the techniques developed for fault tolerance and applies them to provide a novel security tool. Specifically, ReVirt is intended to replay the complete, long-term execution of a computer. To illustrate the difference between these goals, compare the usefulness of checkpoints for each goal. Recovering a backup to a prior point in time can be accomplished either by checkpointing the primary's state periodically or by logging the primary's operations. On the other hand, checkpoints are not sufficient for intrusion analysis because they do not show how the system transitioned between checkpoints; checkpoints can only be used to initialize the replay procedure.

Besides a difference in goals, Hypervisor and ReVirt also differ in several design choices. Because Hypervisor only seeks to restore the backup to the last saved state of the primary, it discards log records after each synchronization point. In contrast, ReVirt enables replay over long periods (e.g. months) of the computer's execution, so it must save all log records since the last checkpoint. Another difference is that Hypervisor defers the delivery of interrupts until the end of a fixed number of instructions (called an epoch), while ReVirt delivers interrupts as soon as they occur (or when the guest kernel re-enables interrupts). Hypervisor also logs more information than ReVirt (e.g. Hypervisor logs disk reads).

There are several virtual machines that are similar to UMLinux. User-Mode Linux [Dike00] shares many of the same goals as UMLinux [Buchacker01]. We chose UMLinux because the virtual machine is contained in a single host process, whereas User-Mode Linux uses a separate host process for each guest application process (this speeds up context switching between guest processes). SimOS's direct-execution mode is also similar to these systems but is targeted at an architecture that is easier to virtualize than the x86 [Rosenblum95].

ReVirt shares a similar philosophy of security logging with S4 [Strunk00]. Both ReVirt and S4 add logging below the target operating system to protect the logging functionality and data from compromised applications and operating systems. ReVirt adds logging to a virtual machine, while S4 adds it to disk drives. The logging in ReVirt captures different information than the logging in S4. ReVirt enables replay of the entire computer's execution, while S4 logs and replays disk activity. ReVirt and S4 save different data to the log (ReVirt saves non-deterministic events, S4 saves disk data), so a comparison of log volume generated will depend on workload.

7. Future work

Our near-term work is to make checkpointing faster and more convenient. We plan to accelerate the disk copy done during checkpointing using copy-on-write. We plan to enable the VMM to checkpoint a running virtual machine by saving and reconstructing the host-kernel state for the virtual-machine process [Plank95].

We also plan to build higher-level analysis tools that leverage ReVirt's ability to replay detailed, long-term executions. Whereas current techniques in computer forensics can only analyze the evidence left behind by careless intruders, ReVirt allows an analyst to watch any intrusion in arbitrary detail.

Finally, we plan to use ReVirt as a building block for new security services. ReVirt's ability to recover to an arbitrary state may enable us to recover a system automatically and to analyze or prevent key events in an attack.

8. Conclusions

ReVirt applies virtual-machine and fault-tolerance techniques to enable a system administrator to replay the long-term, instruction-by-instruction execution of a computer system. Because the target operating system and target applications run within a virtual machine, ReVirt can replay the execution before, during, and after the intruder compromises the system. This capability is especially useful for determining and fixing the damage the intruder inflicted after compromising the system. Because ReVirt logs all non-deterministic events, it can replay non-deterministic attacks and executions, such as those that trigger race conditions. Finally, because ReVirt can replay instruction-by-instruction sequences, it can provide arbitrarily detailed observations about what transpired on the system.

ReVirt adds reasonable time and space overhead. The overhead for virtualization ranges from imperceptible for interactive and CPU-bound applications to 13-58% for kernel-intensive applications. The time overhead of logging ranges from 0-8%, and logging traffic for our workloads can be stored on a single disk for several months.

9. Acknowledgments

We are grateful to the researchers at the University of Erlangen-Nürnberg for writing UMLinux and sharing it with us. In particular, Kerstin Buchacker and Volkmar Sieh helped us understand and use UMLinux. We would also like to thank Brian Noble, Landon Cox, the anonymous reviewers, and our shepherd Jay Lepreau for their helpful feedback. This research was supported in part by National Science Foundation grants CCR-0098229 and CCR-0219085 and by Intel Corporation. Samuel King was supported by a National Defense Science and Engineering Graduate Fellowship.

10. References

[Anderson80] James P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Co., April 1980. Contract 79F296400.

[Ashcraft02] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, May 2002.

[Bellino73] J. Bellino and C. Hans. Virtual Machine or Virtual Operating System? In *Proceedings of the 1973 ACM Workshop on Virtual Computer Systems*, pages 20–29, 1973.

[Bishop96] Matt Bishop and Michael Dilger. Checking for Race Conditions on File Accesses. *USENIX Computing Systems*, 9(2):131–152, 1996.

[Bressoud96] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[Buchacker01] Kerstin Buchacker and Volkmar Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings of the 2001 IEEE Symposium on High Assurance System Engineering (HASE)*, pages 95–105, October 2001.

[CER01a] CERT/CC Security Improvement Modules: Analyze all available information to characterize an intrusion. Technical report, CERT Coordination Center, May 2001.

[CER01b] Linux kernel contains race condition via ptrace/procfs/execve. Technical Report Vulnerability Note VU#176888, CERT Coordination Center, March 2001.

[CER02] CERT/CC Overview Incident and Vulnerability Trends. Technical report, CERT Coordination Center, April 2002.

[Chen01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, May 2001.

[Dike00] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, October 2000.

[Elnozahy02] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[Goldberg74] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–45, June 1974.

[Goldberg96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 1996 USENIX Technical Conference*, July 1996.

[Govil00] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):226--262, August 2000.

[Hon00] Report on the Linux Honeypot Compromise. Technical report, Honeynet Project, November 2000. http://project.honeynet.org/challenge/results/dittrich/evidence.txt.

[Int01] The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. Technical report, Intel Corporation, 2001.

[Karger91] Paul A. Karger, Mary Ellen Zurko, Douglis W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11), November 1991.

[King02] Samuel T. King. Operating System Extensions to Support Host-Based Virtual Machines. Technical Report CSE-TR-465-02, University of Michigan, September 2002.

[LeBlanc87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant

Replay. *IEEE Transactions on Computers*, pages 471--482, April 1987.

[Meushaw00] Robert Meushaw and Donald Simard. NetTop: Commercial Technology in High Assurance Applications. *Tech Trend Notes: Preview of Tomorrow's Information Technologies*, 9(4), September 2000.

[Netzer94] Robert H. B. Netzer and Mark H. Weaver. Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs. In *Proceedings of the 1994 Conference on Programming Language Design and Implementation (PLDI)*, June 1994.

[Plank95] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the Winter 1995 USENIX Conference*, pages 213–224, January 1995.

[Rosenblum95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: the SimOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):34–43, January 1995.

[Russinovich96] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the 1996 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–266, May 1996.

[Strunk00] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[Sugerman01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Technical Conference*, June 2001.

Integrated Resource Management for Cluster-based Internet Services

Kai Shen* kshen@cs.rochester.edu
Hong Tang† htang@cs.ucsb.edu
Tao Yang†§ tyang@cs.ucsb.edu
Lingkun Chu† lkchu@cs.ucsb.edu

* *Dept. of Computer Science, University of Rochester, Rochester, NY 14627*
† *Dept. of Computer Science, University of California, Santa Barbara, CA 93106*
§ *Ask Jeeves/Teoma Technologies, Piscataway, NJ 08854*

Abstract

Client request rates for Internet services tend to be bursty and thus it is important to maintain efficient resource utilization under a wide range of load conditions. Network service clients typically seek services interactively and maintaining reasonable response time is often imperative for such services. In addition, providing differentiated service qualities and resource allocation to multiple service classes can also be desirable at times. This paper presents an integrated resource management framework (part of *Neptune* system) that provides flexible service quality specification, efficient resource utilization, and service differentiation for cluster-based services. This framework introduces the metric of *quality-aware service yield* to combine the overall system efficiency and individual service response time in one flexible model. Resources are managed through a two-level request distribution and scheduling scheme. At the cluster level, a fully decentralized request distribution architecture is employed to achieve high scalability and availability. Inside each service node, an adaptive scheduling policy maintains efficient resource utilization under a wide range of load conditions. Our trace-driven evaluations demonstrate the performance, scalability, and service differentiation achieved by the proposed techniques.

1 Introduction

Previous studies show that the client request rates for Internet services tend to be bursty and fluctuate dramatically [5, 10, 11]. For example, the daily peak-to-average load ratio at Internet search service Ask Jeeves (`www.ask.com`) is typically 3:1 and it can be much higher and unpredictable in the presence of extraordinary events. As another example, the online site of Encyclopedia Britannica (`www.britannica.com`) was taken offline 24 hours after its initial launch in 1999 due to a site overload. Over-provisioning system resources for a service site to accommodate the potential peak will not be cost-effective. As a consequence, it is important to maintain efficient resource utilization for those services under a wide range of load conditions.

Network clients typically seek services interactively and maintaining reasonable response time is imperative. In addition, providing differentiated service qualities and resource allocation to multiple service classes can also be desirable at times, especially when the system is reaching its capacity limit and cannot provide interactive responses to all the requests. Quality of service (QoS) support and service differentiation have been studied extensively in network packet switching with respect to packet delay and connection bandwidth [12, 26, 37]. It is equally important to extend network-level QoS support to endpoint systems where service fulfillment and content generation take place. Those issues are especially critical for cluster-based Internet services in which contents are dynamically generated and aggregated [5, 17, 20, 33, 35].

This paper presents the design and implementation of an integrated resource management framework for cluster-based services. This framework is part of *Neptune* system: a cluster-based software infrastructure for aggregating and replicating partitionable network services [34, 35]. Neptune has been successfully deployed at Internet search engine Ask Jeeves [5] since December 2001. Although cluster-based network services have been widely deployed, we have seen limited research in the literature on comprehensive resource management with service differentiation support. Recent studies on endpoint resource management and QoS support have been mostly focused on single-host systems [1, 2, 6, 7, 8, 27, 39] or clustered systems serving static HTTP content [3, 32]. In comparison, Neptune is intended for clustered services with dynamic service fulfillment or content generation. The work presented in this paper addresses some of the inadequacy of the previous studies and complements them in the following three aspects.

Flexible resource management objectives. Most previous studies used a monolithic metric to measure resource utilization and define QoS constraints. Commonly used ones include system throughput, mean response time, mean stretch factor [41], or the tail distribution of the response time [28]. We introduce a unified quality-aware metric that links the overall system efficiency with individual service response time. To be more specific, we consider the fulfillment of a service request produces certain *quality-aware service yield* depending on the response time. The overall goal of the system is to maximize the aggregate service yield resulting from all requests. As an additional goal, the system supports service differentiation for multiple service classes.

Fully decentralized clustering architecture with quality-aware resource management. Scalability and availability are always overriding concerns for large-scale cluster-based services. Several prior studies relied on centralized components to manage resources for a cluster of replicated servers [3, 10, 32, 41]. In contrast, our framework employs a functionally symmetrical architecture that does not rely on any centralized components. Such a design not only eliminates potential single point of failure in the system, it is also crucial to ensuring smooth and prompt responses to demand spikes and server failures.

Efficient resource utilization under quality constraints. Neptune achieves efficient resource utilization through a two-level request distribution and scheduling scheme. At the cluster level, requests for each service class are evenly distributed to all replicated service nodes without explicit partitioning. Inside each service node, an adaptive scheduling policy adjusts to the runtime load condition and seeks high aggregate service yield at a wide range of load levels. When desired, the service scheduler also provides proportional resource allocation guarantee for specified service classes.

The rest of this paper is organized as follows. Section 2 illustrates a target architecture for this work and then describes our multi-fold resource management objective. Section 3 presents Neptune's two-level request distribution and scheduling architecture. Section 4 illustrates the service scheduling inside each service node. Section 5 presents the system implementation and trace-driven experimental evaluations. Section 6 discusses related work and Section 7 concludes the paper.

2 Targeted Architecture and Resource Management Objective

In this section, we first illustrate the targeted system architecture of this work. Then we introduce the concepts of quality-aware service yield and service yield functions. Through these concepts, service providers can express a variety of quality constraints based on the service response time. Furthermore, using service yield functions and resource allocation guarantees, our framework allows service providers to determine the desired level of service differentiation among multiple service classes.

2.1 Targeted Architecture

Neptune targets cluster-based network services accessible to many users through an intranet or the Internet. Inside those clusters, services are usually partitioned, replicated, aggregated, and then delivered to external clients through protocol gateways. *Partitioning* is introduced when the service processing requirement or data volume exceeds the capacity of a single server node. Service *replication* is commonly employed to improve the system availability and provide load sharing. Partial results may need to be *aggregated* across multiple data partitions or multiple service components before being delivered to external users.

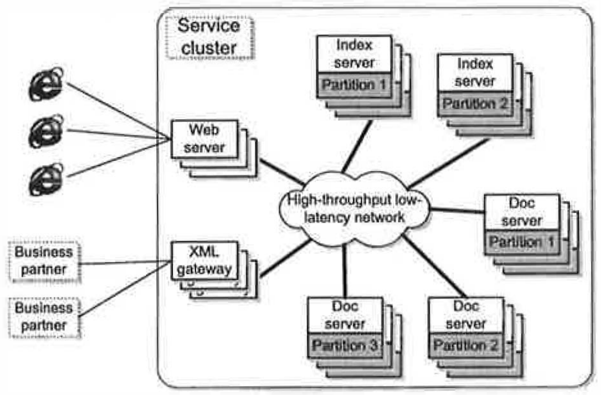


Figure 1: A targeted system architecture: search engine.

Figure 1 uses a prototype search engine to illustrate such a targeted system architecture [5, 18]. In this example, the service cluster delivers search services to consumers and business partners through Web servers and XML gateways. Inside the cluster, the main search tasks are performed on a set of index servers and document servers, both partitioned and replicated. Each search query first arrives at one of the protocol gateways. Then some index servers are contacted to retrieve the identifications of index documents related to the search query. Subsequently some document servers are mobilized to retrieve a short description of these documents and the final results are returned through the original protocol gateway. The resource management work in this study focuses on resources and quality constraints inside the service cluster. Issues related to wide-area network latency or bandwidth is beyond the scope of this paper.

A large-scale service cluster typically consists of multiple groups of replicated service components. We call each replication group a *sub-cluster*. For instance, the replicas for partition 1 of the index servers in Figure 1 form one such sub-cluster. While Neptune supports the construction of multiple sub-clusters, this paper focuses on the resource management within a single sub-cluster. Here we give a brief discussion on the formation of sub-clusters. Each sub-cluster typically hosts a single type of service for modularity and ease of management. This scheme also allows for targeted resource allocation. For instance, machines with large number of CPUs can be allocated for sub-clusters hosting CPU-intensive service components while machines equipped with fast I/O channels can be used for sub-clusters hosting I/O-intensive components. Nonetheless, it is not uncommon to co-locate multiple types of service components in a single replication group to improve resource utilization efficiency.

2.2 Quality-aware Resource Utilization

Most previous studies used a monolithic metric such as system throughput, mean response time, mean stretch factor [41], or the tail distribution of the response time [28] to measure the efficiency of system resource management. We use a more comprehensive metric by conceiving that the fulfillment of a service request provides certain yield depending the response time. This yield, we call *quality-aware service yield*, can be linked to the amount of economic benefit or social reach resulting from serving this request in a timely fashion. Both goals of provisioning QoS and efficient resource utilization can be naturally combined as producing high aggregate yield. Furthermore, we consider the service yield resulting from serving each request to be a function of the service response time. The service yield function is normally determined by service providers to give them flexibility in expressing desired service qualities. Let r_1, r_2, $\cdots$, r_k be the response times of the k service accesses completed in an operation period. Let $Y_i()$ represent the service yield function for the ith service access. The goal of our system is to maximize the aggregate yield, i.e.

$$\text{maximize} \sum_{i=1}^{k} Y_i(r_i). \qquad (1)$$

In general, the service yield function can be any monotonically non-increasing function that returns non-negative numbers with non-negative inputs. We give a few examples to illustrate how service providers can use yield functions to express desired service qualities. For instance, the system with the yield function $Y_{\text{throughput}}$ depicted in Figure 2 (A) is intended to achieve high system throughput with a deadline D. In other words, the goal of such a system is to complete as many service accesses as possible with the response time $\leq D$. Similarly, the system with the yield function Y_{resptime} in Figure 2 (B) is designed to achieve low mean response time. Note that the traditional concept of mean response time does not count dropped requests. Y_{resptime} differs from that concept by considering dropped requests as if they are completed in D.

We notice that $Y_{\text{throughput}}$ does not care about the exact response time of each service access as long as it is completed within the deadline. In contrast, Y_{resptime} always reports higher yield for accesses completed faster. As a hybrid version of these two, Y_{hybrid} in Figure 2 (C) produces full yield when the response time is within a pre-deadline D', and the yield decreases linearly thereafter. The yield finally declines to a *drop penalty* C' when the response time reaches the deadline D. This corresponds to the real world scenario that users are generally comfortable as long as a service request is completed in D'. They get more or less annoyed when the service takes longer and they most likely abandon the service after waiting for D. C represents the full yield resulting from a prompt response and the drop penalty C' represents the loss when the service is not completed within the final deadline D. Figure 2 illustrates these three functions. We want to point out that $Y_{\text{throughput}}$ is a special case of Y_{hybrid} when $D' = D$; and Y_{resptime} is also a special case of Y_{hybrid} when $D' = 0$ and $C' = 0$.

2.3 Service Differentiation

Service differentiation is another goal of our multi-fold resource management objective. Service differentiation is based on the concept of service classes. A *service class* is defined as a category of service accesses that obtain the same level of service support. On the other hand, service accesses belonging to different service classes may receive differentiated QoS support. Service classes can be defined based on client identities. For instance,

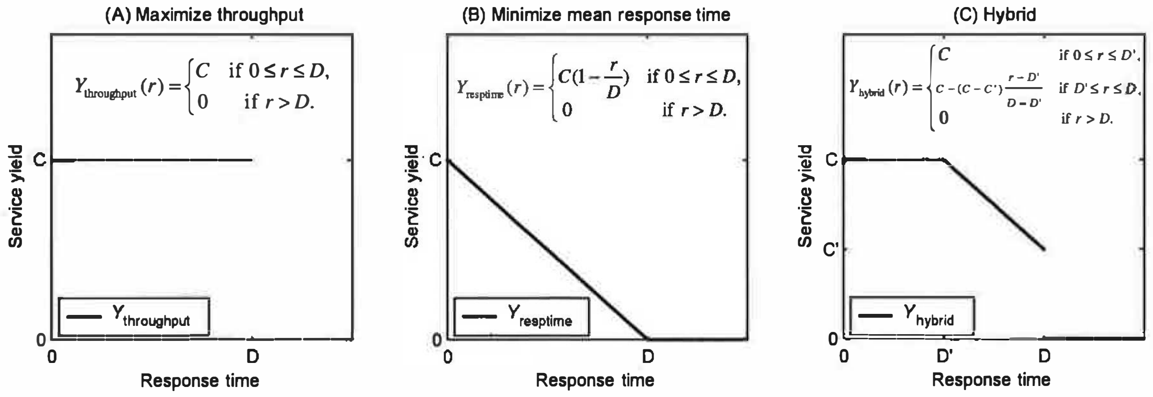


Figure 2: Illustration of service yield functions.

a special group of clients may be configured to receive preferential service support or a guaranteed share of system resources. Service classes can also be defined based on service types or data partitions. For example, a order placement transaction is typically considered more important than a catalog-browsing request.

We provide differentiated services to different service classes on two fronts. First, service classes can acquire differentiated service support by specifying different yield functions. For instance, serving a VIP-class client can be configured to produce higher service yield than serving a regular client. Secondly, each service class can be guaranteed to receive a certain portion of system resources. Most previous service differentiation studies have focused on one of the above two means of QoS support [7, 24, 30, 40]. We believe a combination of them provides two benefits when the system is overloaded: 1) the resource allocation is biased toward high-yield classes for efficient resource utilization; 2) a certain portion of system resources can be guaranteed for each service class, if needed. The second benefit is crucial to preventing starvation for low-priority service classes.

3 Two-level Request Distribution and Scheduling

In our framework, each external service request enters the service cluster through one of the gateways and it is classified into one of the service classes according to rules specified by service providers. Inside the cluster, service components are usually partitioned, replicated, and aggregated to fulfill the request. In this section, we discuss the cluster-level request distribution for a partition group or sub-cluster.

The dynamic partitioning approach proposed in a previous study adaptively partitions all replicas for each sub-cluster into several groups and each group is assigned to handle requests from one service class [41]. We believe such a scheme has a number of drawbacks. First, a cluster-wide scheduler is required to make server partitioning decisions, which is not only a single-point of failure, but also a potential performance bottleneck. Secondly, cluster-wide server groups cannot be repartitioned very frequently, which makes it difficult to respond promptly to changing resource demand. In order to address these problems, Neptune does not explicitly partition server groups. Instead, we employ a symmetrical and decentralized two-level request distribution and scheduling architecture illustrated in Figure 3.

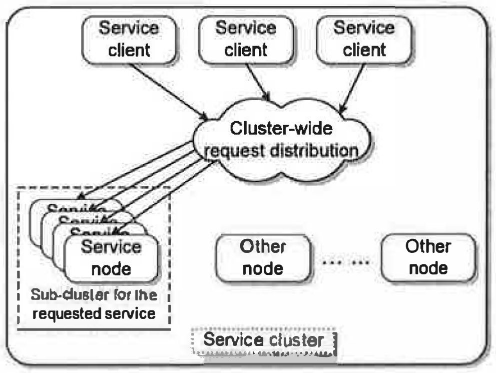


Figure 3: Two-level request distribution and scheduling.

In this scheme, each service node in a sub-cluster can process requests from all service classes. The resource management decision is essentially made at two levels. First, each service request is directed to one of the replicated service nodes through the cluster-level request distribution. Upon arriving at the service node, the request is then subject to a node-level service scheduling. At the cluster level, Neptune employs a class-aware load balancing scheme to evenly distribute requests for each class to all replicas. Our load balancing scheme uses a random polling policy that discards slow-responding polls. Under this policy, whenever a client is about to seek a service for a particular service class, it polls a certain number of randomly selected service nodes to obtain the load information. Then it directs the service request to the node with the smallest number of active and queued requests. Nodes that do not respond within a deadline are discarded. This strategy also helps exclude faulty nodes from request distribution. In practice, we use a poll size of 3 in our system. The polling deadline is set to be 10 ms, which is the smallest timeout granularity supported by `select` system call in Linux. Our recent study shows that such a policy is scalable and it performs well for services of small granularities [34]. Inside each service node, Neptune must also deal with the resource allocation across multiple service classes. This is handled by a node-level class-aware scheduling scheme, which will be discussed in Section 4.

An Alternative Approach for Comparison. For the purpose of comparison, we also designed a request distribution scheme based on server partitioning [41]. Server partitioning is adjusted periodically at fixed intervals. This scheme uses the past resource usage to predict the future resource demand and makes different partitioning decisions during system under-load and overload situations.

- When the aggregate demand does not exceed the total system resources, every service class acquires their demanded resource allocation. The remaining resources will be allocated to all classes proportional to their demand.
- When the system is overloaded, in the first round we allocate to each class its resource demand or its resource allocation guarantee, whichever is smaller. Then the remaining resources are allocated to all classes under a priority order. The priority order is sorted by the full yield divided by the mean resource consumption for each class, which can be acquired through offline profiling.

Fractional server allocations are allowed in this scheme. All servers are partitioned into two pools, a dedicated pool and a shared pool. A service class with 2.4 server allocation, for instance, will get two servers from the dedicated pool and acquire 0.4 server allocation from the shared pool through sharing with other classes with fractional allocations.

The length of the adjustment interval should be chosen carefully so that it is not too small to avoid excessive repartitioning overhead and maintain system stability, nor is it too large to promptly respond to demand changes. We choose the interval to be 10 seconds in this paper. Within each allocation interval, service requests are randomly directed to one of the servers allocated to the corresponding service class according to the load balancing policy [34].

4 Node-level Service Scheduling

Neptune employs a multi-queue (one per service class) scheduler inside each node. Whenever a service request arrives, it enters the appropriate queue for the service class it belongs to. When resources become available, the scheduler picks a request for service. The scheduled request is not necessarily at the head of a queue. Figure 4 illustrates such a runtime environment of a service node.

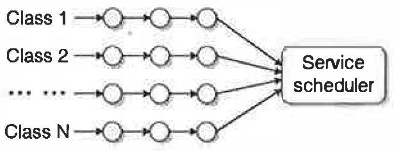


Figure 4: Runtime environment of a service node.

For a service node hosting N service classes: $C_1, C_2, \cdots, C_N$, each class C_k is configured with a service yield function Y_k and optionally a minimum system resource share guarantee g_k, which is expressed as a percentage of total system resources ($\sum_1^N g_k \leq 1$). The goal of the scheduling scheme is to provide the guaranteed system resources for all service classes and schedule the remaining resources to achieve high aggregate service yield. Figure 5 illustrates the framework of our service scheduling algorithm at each scheduling point. In the rest of this section, we will discuss two aspects of the scheduling algorithm: 1) maintaining resource allocation guarantees; and 2) achieving high aggregate service yield.

4.1 Estimating Resource Consumption for Allocation Guarantees

In order to maintain resource allocation guarantees, we need to estimate resource consumption for each service class at each scheduling time. This estimation should be biased toward recent usage to stabilize quickly when the

1. Drop from each queue head those requests that are likely to generate zero or very small yield according to the request arrival time, expected service time and the yield function.
2. Search for the service classes with non-empty request queues that have an estimated resource consumption of less than the guaranteed share. (Section 4.1)
   (a) If found, schedule the one with the largest gap between the resource consumption and the guaranteed share.
   (b) Otherwise, schedule a queued request that is likely to produce high aggregate service yield. (Section 4.2)

Figure 5: The node-level service scheduling algorithm.

actual resource consumption jumps from one level to another. It should not be too shortsighted either in order to avoid oscillations or over-reactions to short-term spikes. Among many possible functions that exhibit those properties, we define the resource consumption for class C_k at time t to be the weighted summation of the resource usage for all class C_k requests completed no later than t. The weight is chosen to decrease exponentially with regard to the elapsed time since the request completion. For each request r, let $ct(r)$ be its completion time and $s(r)$ be its measured resource usage (we will discuss how to measure it in the end of this sub-section), which is known after its completion. Equation 2 defines $u_k(t)$ to be the resource consumption for class C_k at time t. Note that the time in all the following equations is denominated in *seconds*.

$$u_k(t) = \sum_{\{r|r\in C_k \text{ and } ct(r)\leq t\}} \beta^{t-ct(r)} s(r), \qquad 0 < \beta < 1 \tag{2}$$

Another reason for which we choose this function is that it can be incrementally calculated without maintaining the entire service scheduling history. If we adjust $u_k(t)$ at the completion of every request and let t' be the previous calculation time, the resource consumption at time t can be calculated incrementally through Equation 3.

$$u_k(t) = \beta^{t-t'} u_k(t') + \beta^{t-ct(r)} s(r) \tag{3}$$

The selection of β should be careful to maintain the smooth and stable reaction for both short-term spikes and long-term consumption changes. In this paper we empirically choose β to be 0.95. Since we use *second* as the unit of time in those equations, this means a service request completed one second ago carries 95% the weight of a service request completed right now. With the definition of $u_k(t)$, the proportional resource consumption of class C_k can be represented by $\frac{u_k(t)}{\sum_{k=1}^{N} u_k(t)}$. In step 2 of the service scheduling, this proportional consumption is compared with the guaranteed share to search for under-allocated service classes.

This resource consumption estimation scheme is related to the exponentially-weighted moving average (EWMA) filter used as the round-trip time predictor in TCP [14]. It differs from the original EWMA filter in that the weight in our scheme decreases exponentially with regard to the elapsed time instead of the elapsed number of measurement samples. This is more appropriate for estimating resource consumption due to its time-decaying nature.

The detailed measurement of resource consumption $s(r)$ for each request r is application-dependent. Generally speaking, each request can involve mixed CPU and I/O activities and it is difficult to define a generic formula for all applications. Our approach is to let application developers decide how the resource consumption should be accounted. Large-scale service clusters are typically composed of multiple sub-clusters of replicated service components [5]. Each sub-cluster typically hosts a single type of service for modularity and ease of management. Thus requests in the same sub-cluster tend to share similar resource characteristics in terms of I/O and CPU demand and it is not hard in practice to identify a suitable way to measure resource consumptions. In the current implementation, we use the accumulated CPU consumption for a thread or process acquired through Linux `/proc` file system. The effectiveness of this accounting model is demonstrated in our performance evaluation which contains a benchmark involving significant disk I/O.

Using single dimension resources simplifies our resource accounting model. We acknowledge that it could be desirable at times to co-locate CPU-intensive services with I/O-intensive applications to improve resource utilization efficiency. Accounting multi-dimension resources is not directly tackled in this paper. However, we believe a multi-dimensional resource accounting module can be added into our framework to address this issue.

4.2 Achieving High Aggregate Yield

In this section, we examine the policies employed in step 2b of the service scheduling to achieve high aggregated yield. In general, the optimization problem of maximizing the aggregate yield is difficult to solve given the fact that it relies on the advanced knowledge of the resource requirements of pending requests. Even for the offline case in which the cost for each request is known in advance, Karp has shown that the *Job Sequence* problem, which is a restricted case of our optimization problem, is NP-complete [25]. Various priority-based scheduling policies were proposed in real-time database systems to maximize aggregate realized value [22, 23]. Typical policies considered in those systems include Earliest Deadline First scheduling (*EDF*) and Yield or Value-Inflated Deadline scheduling (*YID*). EDF always schedules the queued request with the closest deadline. YID schedules the queued request with the smallest inflated deadline, defined as the relative deadline divided by the expected yield if the request is being scheduled.

Both EDF and YID are designed to avoid or minimize the amount of lost yield. They work well when the system resources are sized to handle transient heavy load [22]. For Internet services, however, the client request rates tend to be bursty and fluctuate dramatically from time to time [5, 10, 11]. Over-provisioning system resources for a service site to accommodate the potential peak will not be cost-effective. During load spikes when systems face sustained arrival demand exceeding the available resources, missed deadlines become unavoidable and the resource management should instead focus on utilizing resources in the most efficient way. This leads us to design a *Greedy* scheduling policy that schedules the request with the lowest resource consumption per unit of expected yield. The Greedy method typically performs well when the system is overloaded. However, it is not optimal because it only maximizes the efficiency for the next scheduled request without considering longer impact of the scheduling decision.

In order to have a scheduling policy that works well at a wide range of load conditions, we further design an *Adaptive* policy that dynamically switches between YID and Greedy scheduling depending on the runtime load condition. The scheduler maintains a 30-second window of recent request dropping statistics. If more than 5% of incoming requests are dropped in the watched window, the system is considered as overload and the Greedy scheduling is employed. Otherwise, the YID scheduling is used.

All the above scheduling policies are priority-based scheduling with different definition of priorities. Table 1 summarizes the priority metrics of the four policies. Three of these policies require a predicted service time and resource consumption for each request at the scheduling time. For the service time, we use an exponentially-weighted moving average of the service time of past requests belonging to the same service class. Resource consumption measurement is application-dependent as we have explained in the previous sub-section. In our current implementation, such a prediction is based on an exponentially-weighted moving average of the CPU consumptions of past requests belonging to the same service class.

Policy	**Priority** (the smaller the higher)
EDF	Relative deadline
YID	Relative deadline divided by expected yield
Greedy	Expected resource consumption divided by expected yield
Adaptive	Dynamically switch between YID (in underload) and Greedy (in overload)

Table 1: Summary of scheduling policies.

5 System Implementation and Experimental Evaluations

Neptune has been implemented on a Linux cluster. In addition to the resource management framework described in this paper, Neptune provides load balancing and replication support for cluster-based services [34, 35]. Application developers can easily deploy services through specifying a set of RPC-like access methods for each service and the clients can access them through a simple programming API. Neptune employs a symmetrical architecture in constructing the service infrastructure. Any node can elect to provide services and seek services from other nodes inside the service cluster. Each external service request is assigned a service class ID upon arriving at any of the gateways. Those requests are directed to one of the replicated service nodes according to the class-aware load balancing scheme. Each server node maintains multiple request queues (one per service class) and a thread pool. To process each service request, a thread is dispatched to invoke the application service component through dynamically-linked libraries. The size of the thread pool is chosen to strike the balance between concurrency and efficiency depending on the application characteristics. The aggregate services are exported to external clients through protocol gateways. Neptune was subsequently ported to Solaris platform. An earlier version of Neptune has been successfully deployed at Internet search engine Ask Jeeves [5] since December 2001. The resource management framework described in this paper, however, has not been incorporated into the production system.

The overall objective of the experimental evaluation is to demonstrate the performance, scalability, and service differentiation achieved by the proposed techniques. In particular, the first goal is to examine the system performance of various service scheduling schemes over a wide range of load conditions. Secondly, we will study the performance and scalability of our cluster-level request distribution scheme. Our third goal is to investigate the system behavior in terms of service differentiation during demand spikes and server failures. All the evaluations were conducted on a rack-mounted Linux cluster with 30 dual 400 MHz Pentium II nodes, each of which contains either 512 MB or 1 GB memory. Each node runs Linux 2.2.15 and has two 100 Mb/s Ethernet interfaces. The cluster is connected by a Lucent P550 Ethernet switch with 22 Gb/s backplane bandwidth.

5.1 Evaluation Workloads

Our evaluation studies are based on two service workloads. The first service is a *Differentiated Search* service based on an index search component from Ask Jeeves search. This service takes in a group of encoded query words; checks a memory mapped index database; and returns a list of URLs that are relevant to input query words. The index database size is around 2.5 GB at each node and it cannot completely fit in memory. The mean service time for this service is around 250 ms in our testbed when each request is served in a dedicated environment.

Differentiated Search distinguishes three classes of clients, representing Gold, Silver, and Bronze memberships. We let the request composition for these three classes be 10%, 30%, 60% respectively. The yield functions of these service classes can be one of the three forms that we described in Section 2.2, i.e. $Y_{\text{throughput}}()$, $Y_{\text{resptime}}()$, or $Y_{\text{hybrid}}()$. In each case, the shapes of the yield functions for three service classes are the same other than the magnitude. We determine the ratio of such magnitudes to be 4:2:1 meaning that processing a Gold request yields four times as much as a Bronze request at the same response time. The deadline D is set to be 2 seconds. In the case of $Y_{\text{hybrid}}()$, the drop penalty C' is set to be half of the full yield and the pre-deadline D' is set to be half of the absolute deadline D. Figure 6 illustrates the yield functions when they are in each one of the three forms.

The request arrival intervals and the query words for the three Differentiated Search service classes are based on a one-week trace we collected at Ask Jeeves search via one of its edge Web servers. The request distribution among the edge Web servers are conducted by a balancing switch according to the "least connections" policy. Note that this trace only represents a fraction of the complete Ask Jeeves traffic during the trace collection period. Figure 7 shows the total and non-cached search rate of this trace. The search engine employs a query cache to directly serve those queries that have already been served before and cached. We are only concerned with non-cached requests in our evaluation because only those requests invoke the index search component. We use the peak-time portion of Tuesday, Wednesday, and Thursday's traces to drive the workload for Gold, Silver, and Bronze classes respectively. For each day, the peak-time portion we choose is the 7-hour period from 11 am to 6pm EST. The statistics of these three traces are listed in Table 2. Note that the arrival intervals of these traces may be scaled when necessary to generate workloads at various demand levels during our evaluation.

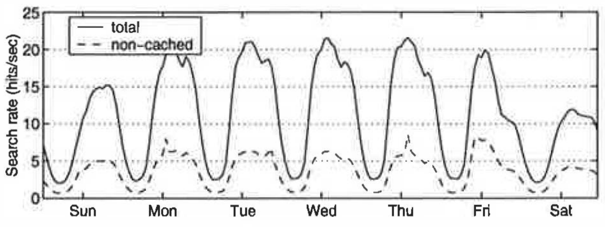


Figure 7: Search requests to Ask Jeeves search via one of its edge web servers (January 6-12, 2002).

	Number of accesses	Arrival interval	
	Total (non-cached)	Mean	Std-dev
Gold	507,202 (154,466)	161.3ms	164.3ms
Silver	512,227 (151,827)	166.0ms	169.5ms
Bronze	517,116 (156,214)	161.3ms	164.7ms

Table 2: Statistics of evaluation traces.

The three service classes in Differentiated Search are based on the same service type and thus have the same average resource consumption. The second service we constructed for the evaluation is designed to have different resource consumption for each service class, representing services differentiated on their types. This service, we call *Micro-benchmark*, is based on a CPU-spinning micro-benchmark. It contains three service classes with the same yield functions as the Differentiated Search service. The mean service times of the three classes are 400 ms, 200 ms, and 100 ms respectively. We use Poisson process arrivals and exponentially distributed service times for the Micro-benchmark service. Several previous studies on Internet connections and workstation clusters suggested that both the HTTP inter-arrival time distribution and the service time distribution exhibit high variance, thus are better modeled by

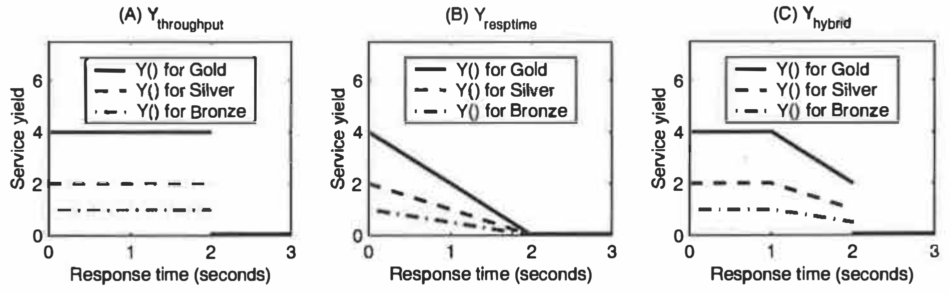


Figure 6: Service yield functions in evaluation workloads.

Lognormal, Weibull, or Pareto distributions [15, 21]. We choose exponentially-distributed arrival intervals and service times for the following reasons. First, a primary cause for the high variance of HTTP arrival intervals is the proximity of the HTTP request for the main page and subsequent requests for embedded objects or images. However, if we only consider resource-intensive service requests which requires dynamic content generation, HTTP requests for embedded objects are not counted. Secondly, the service time distribution tends to have a low variance for services of the same type. Our analysis on the Ask Jeeves trace shows that those distributions have similar variances as an exponentially distributed sample would have.

5.2 Evaluation on Node-level Scheduling and Service Differentiation

In this section, we study the performance of four service scheduling policies (EDF, YID, Greedy and Adaptive) and their impact on service differentiation. The performance metric we use in this study is *LossPercent* [22], which is computed as

$$\text{LossPercent} = \frac{\text{OfferedYield} - \text{RealizedYield}}{\text{OfferedYield}} \times 100\%$$

OfferedYield is the aggregated full yield of all arrived requests and *RealizedYield* is the amount of yield realized by the system. We choose the loss percentage as the performance metric because this metric is effective in illustrating performance difference in both system underload and overload situations. In comparison, the actual rate (aggregate service yield in this case) is not as illustrative as the loss percentage when the system load is below the saturation point.

Figure 8 shows the performance of scheduling policies on Differentiated Search with 16 replicated servers. The experiments were conducted for all three forms of yield functions: $Y_{\text{throughput}}()$, $Y_{\text{resptime}}()$, and $Y_{\text{hybrid}}()$. Figure 9 shows the performance of scheduling policies on Microbenchmark with 16 servers. Only the result for yield functions in $Y_{\text{hybrid}}()$ form is shown to save space. In each case, we show the performance results with a varying arrival demand of up to 200% of the available resources. The demand level cannot simply be the mean arrival rate times the mean service time due to various system overhead. We probe the maximum arrival rate such that more than 95% of all requests are completed within the deadline under EDF scheduling. Then we consider the request demand is 100% at this arrival rate. The desired demand level is then achieved by scaling the request arrival intervals. The performance results are separated into the under-load (arrival demand $\leq$ 100%) and overload (arrival demand $\geq$ 100%) situations. We

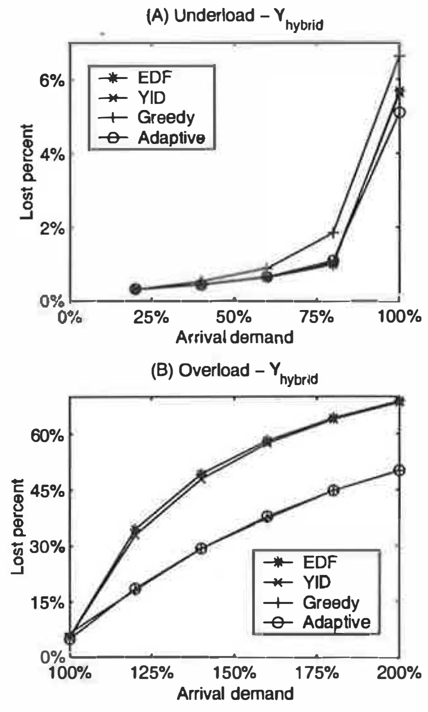


Figure 9: Performance of scheduling policies on Microbenchmark (16 servers).

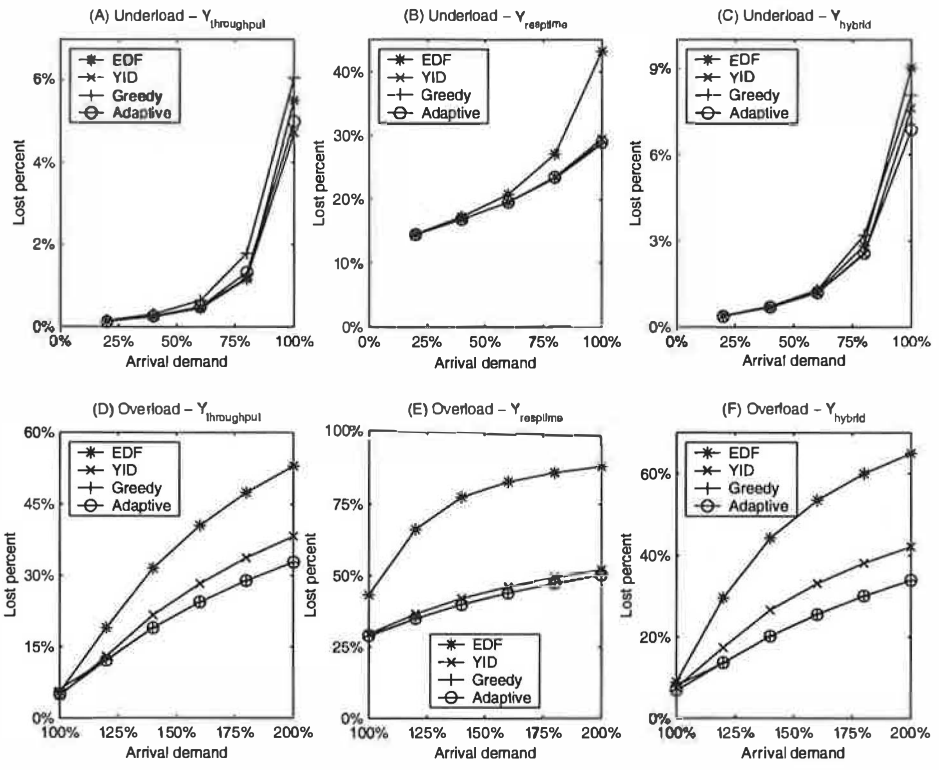


Figure 8: Performance of scheduling policies on Differentiated Search (16 servers).

employ no minimum resource guarantee for both services to better illustrate the comparison on the aggregate yield. From these results, we observe that YID outperforms Greedy by up to 49% when the system is underloaded and Greedy performs up to 39% better during system overload. The Adaptive policy is able to dynamically switch between YID and Greedy policies to achieve good performance on all studied load levels.

To further understand the performance difference among the scheduling policies and the impact on service differentiation, Figure 10 lists the per-class performance breakdown for Differentiated Search service with $Y_{\text{hybrid}}()$ yield functions under 200% arrival demand. For the per-class response time, we show both the mean values and the 95th percentile values. We choose a high arrival demand (200%) for this experiment because service differentiation is more critical at higher load. Extraordinary events can cause such severe system overload and shorter-term spikes can be more widespread in practice. From Figure 10, we observe that all four policies achieve similar aggregate throughput, however, Greedy and Adaptive policies complete more requests of higher-priority classes, representing more efficient resource utilization. In terms of the mean response time, Greedy and Adaptive policies complete requests with shorter mean response time, representing better quality for completed requests.

5.3 Evaluation on Request Distribution across Replicated Servers

Figure 11 illustrates our evaluation results on two request distribution schemes: class-aware load balancing (used in Neptune) and server partitioning. For each service, we show the aggregate service yield of up to 16 replicated servers under slight under-load (75% demand), slight overload (125% demand), and severe overload (200% demand). The Adaptive scheduling policy is used in each server for those experiments. The aggregate yield shown in Figure 11 is normalized to the Neptune yield under 200% arrival demand. Our result shows that both schemes exhibit good scalability, which is attributed to our underlying load balancing strategy, the random-polling policy that discards slow-responding polls [34]. In comparison, Neptune produces up to 6% more yield than server partitioning under high demand. This is because Neptune allows the whole cluster-wide load balancing for all service classes while server partitioning restricts the scope of load balancing to the specific server partition for the corresponding service class, which affects the load balancing performance.

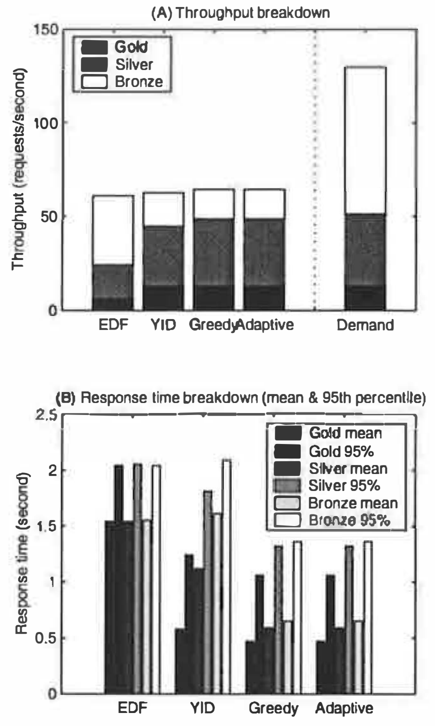


Figure 10: Per-class performance breakdown of Differentiated Search at 200% arrival demand.

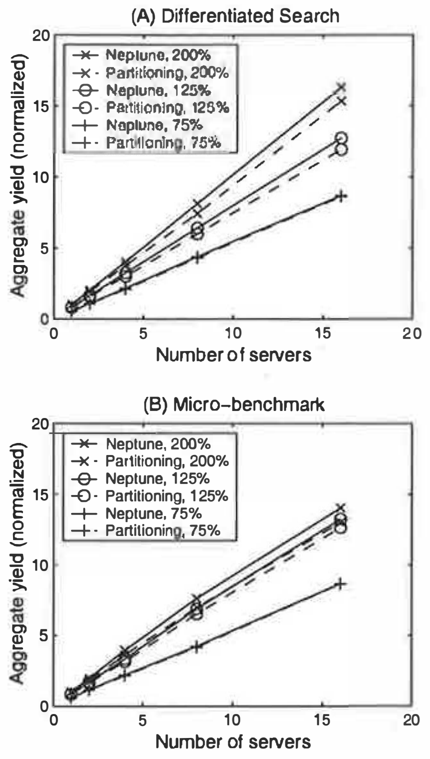


Figure 11: Performance and scalability of request distribution schemes.

5.4 Service Differentiation during Demand Spikes and Server Failures

In this section, we study the service differentiation during demand spikes and server failures. We use the Differentiated Search service with 20% resource guarantee for each class. In order to produce constantly controllable demand levels, we altered this service to generate fixed interval request arrivals. Figure 12 illustrates the system behavior of such a service under Neptune and server partitioning approaches in a 16-server configuration. For each service class, we show the resource demand and the resource allocation, measured in two-second intervals, over a 300-second period.

Initially the total demand is 100% of the available resources, with 10%, 30%, and 60% of which belong to Gold, Silver, and Bronze class respectively. Then there is a demand spike for the Silver class between time 50 and time 150. We observe that Neptune promptly responds to the demand spike by allocating more resources to meet high-priority Silver class demand and dropping some low-priority Bronze class requests. This shift stops when Bronze class resource allocation drops to around 20% of total system resources, which is its guaranteed share. We also see the resource allocations for Silver and Bronze class quickly stabilize when they reach new allocation levels. In comparison, the server partitioning scheme responds to this demand spike in a slower pace because it cannot adjust to immediate demand changes until the next allocation interval. We also observe that the resource allocation for the highest-priority Gold class is isolated from this demand spike under both schemes.

At time 200, one server (allocated to the Gold class under server partitioning) fails and it recovers at time 250. Immediately after the server failure, we see a deep drop of Gold class resource allocation for about 10 seconds under server partitioning. This is again because it cannot adjust to immediate resource change until the next allocation interval. In comparison, Neptune exhibits much smoother behavior because losing any one server results in a proportional loss of resources for each class. Also note that the loss of a server reduces the available resources, which increases the relative demand to the available resources. This effectively results in another resource shortage. The system copes with it by maintaining enough allocation to Gold and Silver classes while dropping some Bronze class requests.

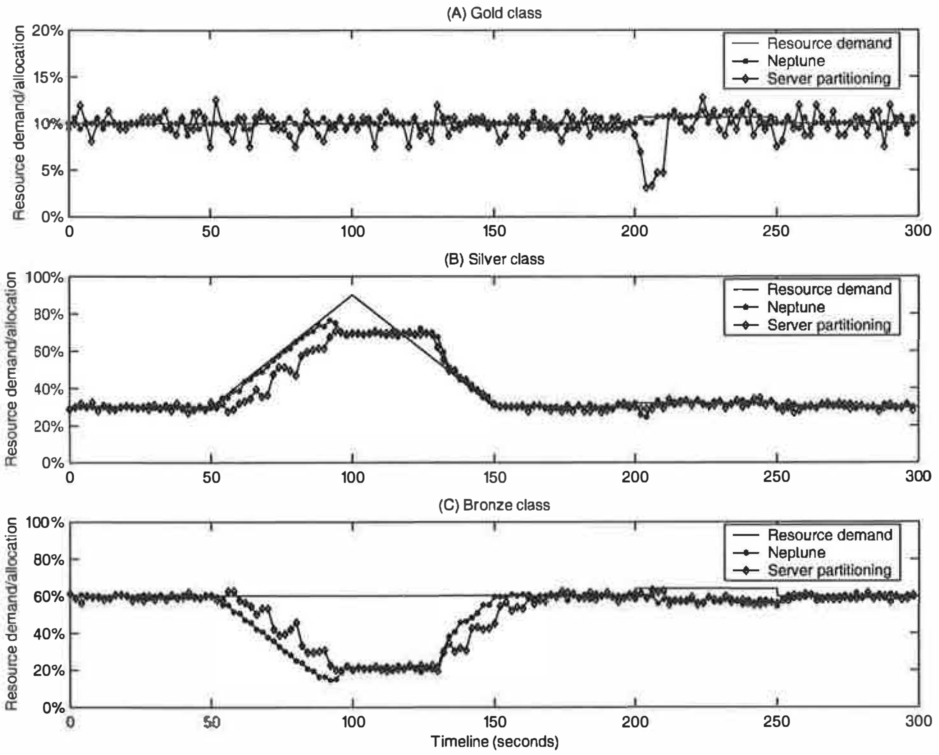


Figure 12: System behavior during demand spike and server failure with 16 servers. Differentiated Search with 20% resource guarantee for each class is used. One server (allocated to the Gold class under server partitioning) fails at time 200 and it recovers at time 250.

6 Related Work

Software infrastructure for clustered services. Previous studies have addressed the scalability and availability issues in providing software infrastructure for cluster-based network services [17, 20, 35]. In particular, TACC employs a two-tier architecture in which the service components called "workers" run on different backends while accesses to workers are controlled by frontends [17]. Our work in this paper complements these studies by proposing an integrated resource management framework that addresses quality specification, efficient resource utilization under quality constraints, and service differentiation support. There are a number of recent studies on replication support for services with frequent updates on persistent service data [19, 33, 35]. The focus of these studies is to support replica consistency in addition to the existing requirements of scalability and availability. The resource management framework proposed in this paper has only been evaluated with read-only workload. The impact of update-intensive workload remains to be addressed in the future.

Quality-of-service support and service differentiation. The importance of providing QoS support and service differentiation has been recognized in the networking community and the focuses of these studies is network bandwidth allocation and packet delay [26, 37]. The methods for ensuring bandwidth usage include delaying or dropping user requests [12, 27, 32] or reducing service qualities [1, 9]. Recent studies on endpoint resource management and QoS support have been mostly focused on single-host systems [1, 2, 6, 7, 8, 27, 39] or clustered systems serving static HTTP content [3, 32]. In comparison, Neptune focuses on achieving efficient resource utilization and providing service differentiation for cluster-based services in which contents are dynamically generated and aggregated. Recent advances in OS research have developed approaches to provide QoS support at OS kernel level [6, 8, 13, 29, 36, 39]. Our work can be enhanced by those studies to support hard QoS guarantees and service differentiation at finer granularities.

The concept of service quality in this resource management framework refers to only the service response time. Service response time is important for many applications such that the proposed techniques can be widely applied. However, we acknowledge that service quality can have various application-specific additional dimensions. For instance, the partial failure in a partitioned search database results in a loss of harvest [16]. Further work is needed to address additional application-specific service qualities.

Resource management for clustered services. A large body of work has been done in request distribution and resource management for cluster-based server systems [3, 4, 10, 31, 38, 41]. In particular, demand-driven service differentiation (DDSD) provides a dynamic server partitioning approach to differentiating services from different service classes [41]. Similar to a few other studies [3, 10], DDSD supports service differentiation in the aggregate allocation for each service class. In comparison, this paper presents a decentralized architecture to achieve scalability while deploying quality-aware resource management.

Locality-aware request distribution. Previous study has proposed locality-aware request distribution (LARD) to exploit application-level data locality for Web server clusters [31]. Our work does not explicitly consider data locality because many applications are not locality-sensitive. For example, the critical working set in many Ask Jeeves service components are designed to fit into the system memory. Over-emphasizing on application-level service characteristics may thus limit the applicability of our framework. Nonetheless, it will be a valuable future work to incorporate locality-aware heuristics into our cluster-level request distribution and evaluate its impact on various applications.

Service scheduling. Deadline scheduling, proportional-share resource scheduling, and value-based scheduling have been studied in both real-time systems and general-purpose operating systems [7, 22, 23, 24, 30, 36, 40]. Client request rates for Internet services tend to be bursty and fluctuate dramatically from time to time [5, 10, 11]. Delivering satisfactory user experience is important during load spikes. Based on an adaptive scheduling approach and a resource consumption estimation scheme, the service scheduling in Neptune strives to achieve efficient resource utilization under quality constraints and provide service differentiation.

7 Concluding Remarks

This paper presents the design and implementation of an integrated resource management framework for cluster-based network services. This framework is flexible in allowing service providers to express desired service qualities based on the service response time. At the cluster level, a scalable decentralized request distribution architecture ensures prompt and smooth response to service demand spikes and server failures. Inside each node, an adaptive multi-queue scheduling scheme is employed to achieve efficient resource utilization under quality constraints and provide service differentiation. Our trace-driven evaluations show that the proposed techniques can efficiently utilize system resources under quality constraints and provide service differentiation. Comparing with a previously proposed dynamic server partitioning approach, the evaluations also show that our system responds more promptly to demand spikes and behaves more smoothly during server failures.

Acknowledgment: This work was supported in part by NSF CCR-9702640, EIA-0080134, ACIR-0082666 and 0086061. We would like to thank Anurag Acharya, Josep Blanquer, Apostolos Gerasoulis, Klaus Schauser, our shepherd Jim Gray, and the anonymous referees for their valuable comments and help.

Project Web site:
`www.cs.ucsb.edu/projects/neptune`

References

[1] T. F. Abdelzaher and N. Bhatti. Web Server QoS Management by Adaptive Content Delivery. In *International Workshop on Quality of Service*, London, UK, June 1999.

[2] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing Differentiated Levels of Service in Web Content Hosting. In *Proc. of SIGMETRICS Workshop on Internet Server Performance*, Madison, WI, June 1998.

[3] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proc. of the 2000 ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 90–101, Santa Clara, CA, June 2000.

[4] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Services. In *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[5] Ask jeeves search. http://www.ask.com.

[6] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.

[7] N. Bhatti and R. Friedrich. Web Server Support for Tiered Services. *IEEE Network*, 13(5):64–71, September 1999.

[8] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proc. of USENIX Annual Technical Conf.*, pages 235–246, Orleans, LA, June 1998.

[9] S. Chandra, C. S. Ellis, and A. Vahdat. Differentiated Multimedia Web Services Using Quality Aware Transcoding. In *Proc. of IEEE INFOCOM'2000*, Tel-Aviv, Israel, March 2000.

[10] J. S. Chase, D. C. Anderson, P. N. Thakar, and A. M. Vahdat. Managing Energy and Server Resources in Hosting Centers. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[11] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, 1997.

[12] C. Dovrolis, D. Stiliadis, and P. Ramanathan. Proportional Differentiated Services: Delay Differentiation and Packet Scheduling. In *Proc. of ACM SIGCOMM'99*, pages 109–120, Cambridge, MA, August 1999.

[13] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 1996.

[14] J. Postel Ed. Transmission Control Protocol Specification. SRI International, Menlo Park, CA, September 1981. RFC-793.

[15] A. Feldmann. Characteristics of TCP Connection Arrivals. Technical report, AT&T Labs Research, 1998.

[16] A. Fox and E. A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Proc. of HotOS-VII*, Rio Rico, AZ, March 1999.

[17] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint Malo, October 1997.

[18] Google search. http://www.google.com.

[19] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proc. of the 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[20] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler. The MultiSpace: An Evolutionary Platform for Infrastructural Services. In *Proc. of the USENIX Annual Technical Conf.*, Monterey, CA, June 1999.

[21] M. Harchol-Balter and A. B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[22] J. R. Haritsa, M. J. Carey, and M. Livny. Value-Based Scheduling in Real-Time Database Systems. *VLDB Journal*, 2:117–152, 1993.

[23] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham. Experimental Evaluation of Real-Time Transaction Processing. In *Proc. of the Tenth IEEE Real-Time System Symposium*, pages 144–153, Santa Monica, CA, 1989.

[24] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 198–211, Saint-Malo, France, October 1997.

[25] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, March 1972.

[26] J. Kurose. Open Issues and Challenges in Providing Quality of Service Guarantees in High-Speed Networks. *ACM Computer Communication Review*, 23(1):6–15, 1993.

[27] K. Li and S. Jamin. A Measurement-Based Admission-Controlled Web Server. In *Proc. of IEEE INFOCOM'2000*, pages 651–659, Tel-Aviv, Israel, March 2000.

[28] Z. Liu, M. S. Squillante, and J. L. Wolf. On Maximizing Service-Level-Agreement Profits. In *Proc. of 3rd ACM Conference on Electronic Commerce*, pages 14–17, Tampa, FL, October 2001.

[29] J. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proc. of USENIX Annual Technical Conf.*, San Diego, CA, January 1996.

[30] S. Nagy and A. Bestavros. Admission Control for Soft-Deadline Transactions in ACCORD. In *Proc. of IEEE Real-Time Technology and Applications Symposium*, pages 160–165, Montreal, Canada, June 1997.

[31] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proc. of the ACM 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, San Jose, CA, October 1998.

[32] R. Pandey, J. F. Barnes, and R. Olsson. Supporting Quality of Service in HTTP Servers. In *Proc. of 17th ACM Symposium on Principles of Distributed Computing*, pages 247–256, Puerto Vallarta, Mexico, June 1998.

[33] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability, and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 1–15, Charleston, SC, December 1999.

[34] K. Shen, T. Yang, and L. Chu. Cluster Load Balancing for Fine-grain Network Services. In *Proc. of International Parallel & Distributed Processing Symposium*, Fort Lauderdale, FL, April 2002.

[35] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems*, pages 197–208, San Francisco, CA, March 2001.

[36] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proc. of 3rd USENIX Operating Systems Design and Implementation Symposium*, New Orleans, LA, February 1999.

[37] I. Stoica and H. Zhang. LIRA: An Approach for Service Differentiation in the Internet. In *Proc. of Nossdav*, June 1998.

[38] D. G. Sullivan and M. I. Seltzer. Isolation with Flexibility: A Resource Management Framework for Central Servers. In *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[39] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *Proc. of USENIX Annual Technical Conf.*, Boston, MA, June 2001.

[40] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. of USENIX Operating Systems Design and Implementation Symposium*, pages 1–11, Monterey, CA, November 1994.

[41] H. Zhu, H. Tang, and T. Yang. Demand-driven Service Differentiation for Cluster-based Network Servers. In *Proc. of IEEE INFOCOM'2001*, Anchorage, AK, April 2001.

Resource Overbooking and Application Profiling in Shared Hosting Platforms *

Bhuvan Urgaonkar, Prashant Shenoy and **Timothy Roscoe**†

Department of Computer Science,
University of Massachusetts
Amherst MA 01003
{bhuvan, shenoy}@cs.umass.edu

†Intel Research at Berkeley
2150 Shattuck Avenue Suite 1300
Berkeley CA 94704
troscoe@intel-research.net

Abstract

In this paper, we present techniques for provisioning CPU and network resources in shared hosting platforms running potentially antagonistic third-party applications. The primary contribution of our work is to demonstrate the feasibility and benefits of overbooking resources in shared platforms, to maximize the platform yield*: the revenue generated by the available resources. We do this by first deriving an accurate estimate of application resource needs by profiling applications on dedicated nodes, and then using these profiles to guide the placement of application components onto shared nodes. By overbooking cluster resources in a controlled fashion, our platform can provide performance guarantees to applications even when overbooked, and combine these techniques with commonly used QoS resource allocation mechanisms to provide application isolation and performance guarantees at run-time. When compared to provisioning based on the worst-case, the efficiency (and consequently revenue) benefits from controlled overbooking of resources can be dramatic. Specifically, experiments on our Linux cluster implementation indicate that overbooking resources by as little as 1% can increase the utilization of the cluster by a factor of two, and a 5% overbooking yields a 300-500% improvement, while still providing useful resource guarantees to applications.*

1 Introduction and Motivation

Server clusters built using commodity hardware and software are an increasingly attractive alternative to traditional large multiprocessor servers for many applications, in part due to rapid advances in computing technologies and falling hardware prices.

This paper addresses challenges in the design of a type of server cluster we call a *shared hosting platform*. This can be contrasted with a *dedicated hosting platform*, where either the entire cluster runs a single application (such as a web search engine), or each individual processing element in the cluster is dedicated to a single application (as in the "managed hosting" services provided by some data centers). In contrast, shared hosting platforms run a large number of different third-party applications (web servers, streaming media servers, multi-player game servers, e-commerce applications, etc.), and the number of applications typically *exceeds the number of nodes in the cluster*. More specifically, each application runs on a subset of the nodes and these subsets may overlap. Whereas dedicated hosting platforms are used for many niche applications that warrant their additional cost, economic reasons of space, power, cooling and cost make shared hosting platforms an attractive choice for many application hosting environments.

Shared hosting platforms imply a business relationship between the *platform provider* and *application providers*: the latter pay the former for resources on the platform. In return, the platform provider gives some kind of guarantee of resource availability to applications [17]. The central challenge to the platform provider is thus one of resource management: the ability to reserve resources for individual applications, the ability to isolate applications from other misbehaving or overloaded applications, and the ability to provide performance guarantees to applications.

Arguably, the widespread deployment of shared hosting platforms has been *hampered* by the lack of effective resource management mechanisms that meet these requirements. Most hosting platforms in use today adopt one of two approaches.

The first avoids resource sharing altogether by employing a dedicated model. This delivers useful resources to application providers, but is expensive in machine re-

*Portions of this research were done when Timothy Roscoe was a researcher at Sprint ATL and Bhuvan Urgaonkar was a summer intern at Sprint ATL. This research was supported in part by NSF grants CCR-9984030, EIA-0080119 and a gift from Sprint Corporation.

sources. The second approach shares resources in a best-effort manner among applications, which consequently receive no resource guarantees. While this is cheap in resources, the value delivered to application providers is limited. Consequently, both approaches imply an economic disincentive to deploy viable hosting platforms.

Recently, several resource management mechanisms for shared hosting platforms have been proposed [1, 3, 7, 23]. This paper reports work performed in this context, but with two significant differences in goals.

Firstly, we seek from the outset to support a diverse set of potentially antagonistic network services simultaneously on a platform. The services will therefore have heterogeneous resource requirements; web servers, continuous media processors, and multiplayer game engines all make different demands on the platform in terms of resource bandwidth and latency. We evaluate our system with such a diverse application mix.

Secondly, we aim to support resource management policies based on *yield management* techniques such as those employed in the airline industry [19]. Yield management is driven by the business relationship between a platform provider and many application providers, and results in different short-term goals. In traditional approaches the most important aim is to satisfy all resource contracts while making efficient use of the platform. Yield management by contrast is concerned with ensuring that as much of the available resource as possible is used to generate revenue, rather than being utilized "for free" by a service (since it would otherwise be idle).

An analogy with air travel may clarify the point: instead of trying to ensure that every ticketed passenger gets to board their chosen flight, we try to ensure that no plane takes off with an empty seat (which is achieved by overbooking seats).

An immediate consequence of this goal is our treatment of "flash crowds"—a shared hosting platform should react to an unexpected high demand on an application only if there is an economic incentive for doing so. That is, the platform should allocate additional resources to an application only if it enhances revenue. Further, any increase in resource allocation of an application to handle unexpected high demands should not be at the expense of contract violations for other applications, since this is economically undesirable.

1.1 Research Contributions

The contribution of this paper is threefold. First, we show how the resource requirements of an application can be derived using online profiling and modeling. Second, we demonstrate the efficiency benefits to the platform provider of *overbooking* resources on the platform, and how this can be usefully done without adversely impacting the guarantees offered to application providers. Thirdly, we show how untrusted and/or mutually antagonistic applications in the platform can be isolated from one another. The rest of this section presents these contributions in detail.

Automatic derivation of QoS requirements: We discuss techniques for empirically deriving an application's resource needs. The effectiveness of a resource management technique is crucially dependent on the ability to reserve appropriate resources for each application—overestimating an application's resource needs can result in idling of resources, while underestimating them can degrade application performance. Consequently a shared hosting platform can significantly enhance its utility to users by automatically deriving the QoS requirements of an application. Automatic derivation of QoS requirements involves (i) monitoring an application's resource usage, and (ii) using these statistics to derive QoS requirements that conform to the observed behavior.

We employ kernel-based profiling mechanisms to empirically monitor an application's resource usage and propose techniques to derive QoS requirements from this observed behavior. We then use these techniques to experimentally profile several server applications such as web, streaming, game, and database servers. Our results show that the bursty resource usage of server applications makes it feasible to extract statistical multiplexing gains by overbooking resources on the hosting platform.

Revenue maximization through overbooking: We discuss resource overbooking techniques and application placement strategies for shared hosting platforms. Provisioning cluster resources solely based on the worst-case needs of an application results in low average utilization, since the average resource requirements of an application are typically smaller than its worst case (peak) requirements, and resources tend to idle when the application does not utilize its peak reserved share. In contrast, provisioning a cluster based on a high *percentile* of the application needs yields statistical multiplexing gains that significantly increase the average utilization of the cluster at the expense of a small amount of overbooking, and increases the number of applications that can be supported on a given hardware configuration.

A well-designed hosting platform should be able to provide performance guarantees to applications even when overbooked, with the proviso that this guarantee is now probabilistic (for instance, an application might be provided a 99% guarantee (0.99 probability) that its resource needs will be met). Since different applications have different tolerance to such overbooking (e.g., the latency requirements of a game server make it less tolerant to violations of performance guarantees than a web server), an overbooking mechanism should take into account diverse application needs.

We demonstrate the feasibility and benefits of overbooking resources in shared hosting platforms, and propose techniques to overbook (i.e. under-provision) resources in a controlled fashion based on application resource needs. Although such overbooking can result in transient overloads where the aggregate resource demand temporarily exceeds capacity, our techniques limit the chances of transient overload of resources to predictably rare occasions, and provide useful performance guarantees to applications in the presence of overbooking. The techniques we describe are general enough to work with many commonly used OS resource allocation mechanisms.

Placement and isolation of antagonistic applications: We describe an additional aspect of the resource management problem: placement and isolation of antagonistic applications. We assume that third-party applications may be antagonistic cannot be trusted by the platform, due either to malice or bugs. Our work demonstrates how untrusted third-party applications can be isolated from one another in shared hosting platforms in two ways. Local to a machine, each processing node in the platform employs resource management techniques that "sandbox" applications by restricting the resources consumed by an application to its reserved share. Globally, we present automated placement techniques that allow a platform provider to exert sufficient control over the placement of application components onto nodes in the cluster, since manual placement of applications is unfeasibly complex and error-prone in large clusters.

1.2 System Model and Terminology

The shared hosting platform assumed in our research consists of a cluster of *nodes*, each of which consists of processor, memory, and storage resources as well as one or more network interfaces. Platform nodes are allowed to be heterogeneous with different amounts of these resources on each node. The nodes in the hosting platform are assumed to be interconnected by a high-speed LAN such as gigabit ethernet (see Figure 1). Each cluster node is assumed to run an operating system kernel that supports some notion of quality of service such as reservations or shares. In this paper, we focus on managing CPU and network interface bandwidth in shared hosting platforms. As [1] points out, management of other resources which are inherently temporal in nature, such as disk bandwidth, can be performed by similar mechanisms. Spatial resources, in particular physical memory, present a different challenge. A straightforward approach is to use static partitioning as in [1], although recently more sophisticated approaches have been implemented [5].

We use the term *application* for a complete service running on behalf of an application provider; since an application will frequently consist of multiple distributed components, we use the term *capsule* to refer to the component of an application running on a single node. Each application has at least one capsule, possibly more if the application is distributed. Capsules provide a useful abstraction for logically partitioning an application into subcomponents and for exerting control over the distribution of these components onto different nodes. To illustrate, consider an e-commerce application consisting of a web server, a Java application server and a database server. If all three components need to be colocated on a single node, then the application will consist of a single capsule with all three components. On the other hand, if each component needs to be placed on a different node, then the application should be partitioned into three capsules. Depending on the number of its capsules, each application runs on a subset of the platform nodes and these subsets can overlap with one another, resulting in resource sharing (see Figure 1).

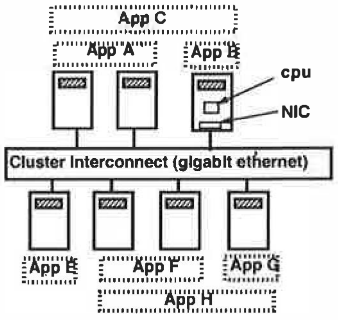


Figure 1: Architecture of a shared hosting platform. Each application runs on one or more nodes and shares resources with other applications.

The rest of this paper is structured as follows. Section 2 discusses techniques for empirically deriving an application's resource needs, while Section 3 discusses our resource overbooking techniques and capsule placement strategies. We discuss implementation issues in Section 4 and present our experimental results in Section 5. Section 6 discusses related work, and finally, Section 7 presents concluding remarks.

2 Automatic Derivation of Application QoS Requirements

The first step in hosting a new application is to derive its resource requirements. While the problem of QoS-aware resource management has been studied extensively in the literature [4, 12, 13], the problem of *how much* resource to allocate to each application has received relatively little attention. In this section, we address this issue by proposing techniques to automatically derive the QoS requirements of an application (the terms resource requirements and QoS requirements are used interchangeably in this paper.) Deriving the QoS requirements is a two step process: (i) we first use profiling techniques to monitor application behavior, and (ii) we then use our empirical measurements to derive QoS requirements that conform to the observed behavior.

2.1 Application QoS Requirements: Definitions

The QoS requirements of an application are defined on a per-capsule basis. For each capsule, the QoS requirements specify the intrinsic rate of resource usage, the variability in the resource usage, the time period over which the capsule desires resource guarantees, and the level of overbooking that the application (capsule) is willing to tolerate. As explained earlier, in this paper, we are concerned with two key resources, namely CPU and network interface bandwidth. For each of these resources, we define the QoS requirements along the above dimensions in an OS-independent manner. In Section 4.1, we show how to map these requirements to various OS-specific resource management mechanisms that have been developed.

More formally, we represent the QoS requirements of an application capsule by a quintuple $(\sigma, \rho, \tau, U, O)$:

Token Bucket Parameters (σ,ρ)*:* We capture the basic resource requirements of a capsule by modeling resource usage as a token bucket (σ, ρ) [22]. The parameter σ denotes the intrinsic rate of resource consumption, while ρ denotes the variability in the resource consumption. More specifically, σ denotes the rate at which the capsule consumes CPU cycles or network interface bandwidth, while ρ captures the maximum burst size. By definition, a token bucket bounds the resource usage of the capsule to $\sigma \cdot t + \rho$ over any interval t.

Period τ*:* The third parameter τ denotes the time period over which the capsule desires guarantees on resource availability. Put another way, the system should strive to meet the QoS requirements of the capsule over each interval of length τ. The smaller the value of τ, the more stringent are the desired guarantees (since the capsule needs to be guaranteed resources over a finer time scale). In particular, for the above token bucket parameters, the capsule requires that it be allocated at least $\sigma \cdot \tau + \rho$ resources every τ time units.

Usage Distribution U*:* While the token bucket parameters succinctly capture the capsule's resource requirements, they are not sufficiently expressive by themselves to denote the QoS requirements in the presence of overbooking. Consequently, we use two additional parameters—U and O—to specify resource requirements in the presence of overbooking. The first parameter U denotes the probability distribution of resource usage. Note that U is a more detailed specification of resource usage than the token bucket parameters (σ, ρ), and indicates the probability with which the capsule is likely to use a certain fraction of the resource (i.e., $U(x)$ is the probability that the capsule uses a fraction x of the resource, $0 \leq x \leq 1$). A probability distribution of resource usage is necessary so that the hosting platform can provide (quantifiable) probabilistic guarantees even in the presence of overbooking.

Overbooking Tolerance O*:* The parameter O is the overbooking tolerance of the capsule. It specifies the probability with which the capsule's requirements may be violated due to resource overbooking (by providing it with less resources than the required amount). Thus, the overbooking tolerance indicates the minimum level of service that is acceptable to the capsule. To illustrate, if $O = 0.01$, the capsule's resource requirements should be met 99% of the time (or with a probability of 0.99 in each interval τ).

In general, we assume that parameters τ and O are specified by the application provider. This may be based on a contract between the platform provider and the application provider (e.g., the more the application provider is willing to pay for resources, the stronger are the provided guarantees), or on the particular characteristics of the application (e.g., a streaming media server requires more stringent guarantees and is less tolerant to violations of these guarantees). In the rest of this section, we show how to derive the remaining three parameters σ, ρ and U using profiling, given values of τ and O.

2.2 Kernel-based Profiling of Resource Usage

Our techniques for empirically deriving the QoS requirements of an application rely on profiling mechanisms that monitor application behavior. Recently, a number of application profiling mechanisms ranging from OS-kernel-based profiling to run-time profiling using specially linked libraries have been proposed.

We use kernel-based profiling mechanisms in the context of shared hosting platforms, for a number of reasons. Firstly, being kernel-based, these mechanisms work with any application and require no changes to the application at the source or binary levels. This is especially important in hosting environments where the platform provider may have little or no access to third-party applications. Secondly, accurate estimation of an application's resource needs requires detailed information about when and how much resources are used by the application at a fine time-scale. Whereas detailed resource allocation information is difficult to obtain using application-level techniques, kernel-based techniques can provide precise information about various kernel events such as CPU scheduling instances and network packet transmissions times.

The profiling process involves running the application on a set of isolated platform nodes (the number of nodes required for profiling depends on the number of capsules).

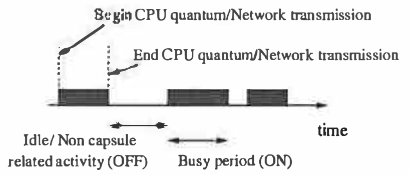

Begin CPU quantum/Network transmission
End CPU quantum/Network transmission
time
Idle/ Non capsule related activity (OFF)
Busy period (ON)

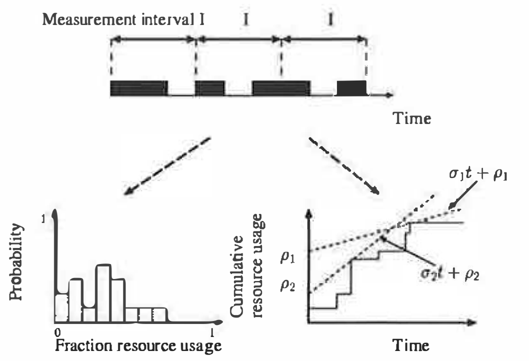
Figure 2: An example of an On-Off trace.

By isolated, we mean that each node runs only the minimum number of system services necessary for executing the application and no other applications are run on these nodes during the profiling process—such isolation is necessary to minimize interference from unrelated tasks when determining the application's resource usage. The application is then subjected to a realistic workload, and the kernel profiling mechanism is used to track its resource usage. It is important to emphasize that the workload used during profiling should be both realistic and representative of real-world workloads. While techniques for generating such realistic workloads are orthogonal to our current research, we note that a number of different workload-generation techniques exist, ranging from trace replay of actual workloads to running the application in a "live" setting, and from the use of synthetic workload generators to the use of well-known benchmarks. Any such technique suffices for our purpose as long as it realistically emulates real-world conditions, although we note that, from a business perspective, running the application "for real" on an isolated machine to obtain a profile may be preferable to other workload generation techniques.

We use the Linux trace toolkit as our kernel profiling mechanism [14]. The toolkit provides flexible, low-overhead mechanisms to trace a variety of kernel events such as system call invocations, process, memory, file system and network operations. The user can specify the specific kernel events of interest as well as the processes that are being profiled to selectively log events. For our purposes, it is sufficient to monitor CPU and network activity of capsule processes—we monitor CPU scheduling instances (the time instants at which capsule processes get scheduled and the corresponding quantum durations) as well as network transmission times and packet sizes. Given such a trace of CPU and network activity, we now discuss the derivation of the capsule's QoS requirements.

2.3 Empirical Derivation of the QoS Requirements

We use the trace of kernel events obtained from the profiling process to model CPU and network activity as a simple On-Off process. This is achieved by examining the time at which each event occurs and its duration and deriving a sequence of busy (On) and idle (Off) periods



(a) Usage distribution (b) Token bucket parameters

Figure 3: Derivation of the usage distribution and token bucket parameters.

from this information (see Figure 2). This trace of busy and idle periods can then be used to derive both the resource usage distribution U as well as the token bucket parameters (σ, ρ).

Determining the usage distribution U: Recall that, the usage distribution U denotes the probability with which the capsule uses a certain fraction of the resource. To derive U, we simply partition the trace into measurement intervals of length $\mathcal{I}$ and measure the fraction of time for which the capsule was busy in each such interval. This value, which represents the fractional resource usage in that interval, is histogrammed and then each bucket is normalized with respect to the number of measurement intervals $\mathcal{I}$ in the trace to obtain the probability distribution U. Figure 3(a) illustrates this process.

Deriving token bucket parameters (σ, ρ): Recall that a token bucket limits the resource usage of a capsule to $\sigma \cdot t + \rho$ over any interval t. A given On-Off trace can have, in general, many (σ, ρ) pairs that satisfy this bound. To intuitively understand why, let us compute the cumulative resource usage for the capsule over time. The cumulative resource usage is simply the total resource consumption thus far and is computed by incrementing the cumulative usage after each ON period. Thus, the cumulative resource usage is a step function as depicted in Figure 3(b). Our objective is to find a line $\sigma \cdot t + \rho$ that bounds the cumulative resource usage; the slope of this line is the token bucket rate σ and its Y-intercept is the burst size ρ. As shown in Figure 3(b), there are in general many such curves, all of which are valid descriptions of the observed resource usage.

Several algorithms that mechanically compute all valid (σ, ρ) pairs for a given On-Off trace have been proposed recently. We use a variant of one such algorithm [22] in our research—for each On-Off trace, the algorithm produces a range of σ values (i.e., $[\sigma_{min}, \sigma_{max}]$) that constitute valid token bucket rates for observed behavior. For each σ within this range, the algorithm also computes the corresponding burst size ρ. Although any pair within this range conforms to the observed behavior, the choice of a particular (σ, ρ) has important practical implications.

Since the overbooking tolerance O for the capsule is given, we can use O to choose a particular (σ, ρ) pair. To illustrate, if $O = 0.05$, the capsule needs must be met 95% of the time, which can be achieved by reserving resources corresponding to the 95^{th} percentile of the usage distribution. Consequently, a good policy for shared hosting platforms is to *pick a σ that corresponds to the $(1 - O) * 100^{th}$ percentile of the resource usage distribution U,* and to pick the corresponding ρ as computed by the above algorithm. This ensures that we provision resources based on a high percentile of the capsule's needs and that this percentile is chosen based on the specified overbooking tolerance O.

2.4 Profiling Server Applications: Experimental Results

In this section, we profile several commonly-used server applications to illustrate the process of deriving an application's QoS requirements. Our experimentally derived profiles not only illustrate the inherent nature of various server application but also demonstrate the utility and benefits of resource overbooking in shared hosting platforms.

The test bed for our profiling experiments consists of a cluster of five Dell PowerEdge 1550 servers, each with a 966 MHz Pentium III processor and 512 MB memory running Red Hat Linux 7.0. All servers runs the 2.2.17 version of the Linux kernel patched with the Linux trace toolkit version 0.9.5, and are connected by 100Mbps Ethernet links to a Dell PowerConnect (model no. 5012) ethernet switch.

To profile an application, we run it on one of our servers and use the remaining servers to generate the workload for profiling. We assume that all machines are lightly loaded and that all non-essential system services (e.g., mail services, X windows server) are turned off to prevent interference during profiling. The parameters τ and $\mathcal{I}$ were both set to 1 sec in all our experimentation. We profile the following server applications in our experiments:

- *Apache web server:* We use the SPECWeb99 benchmark [20] to generate the workload for the Apache web server (version 1.3.24). The SPECWeb benchmark allows control along two dimensions—the number of concurrent clients and the percentage of dynamic (cgi-bin) HTTP requests. We vary both parameters to study their impact on Apache's resource needs.

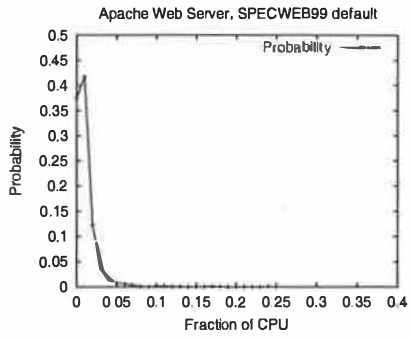
- *MPEG streaming media server:* We use a home-grown streaming server to stream MPEG-1 video files to multiple concurrent clients over UDP. Each client in our experiment requests a 15 minute long variable bit rate MPEG-1 video with a mean bit rate of 1.5 Mb/s. We vary the number of concurrent clients and study its impact on the resource usage at the server.

- *Quake game server:* We use the publicly available Linux Quake server to understand the resource usage of a multi-player game server; our experiments use the standard version of Quake I—a popular multi-player game on the Internet. The client workload is generated using a bot—an autonomous software program that emulates a human player. We use the publicly available "terminator" bot to emulate each player; we vary the number of concurrent players connected to the server and study its impact on the resource usage.

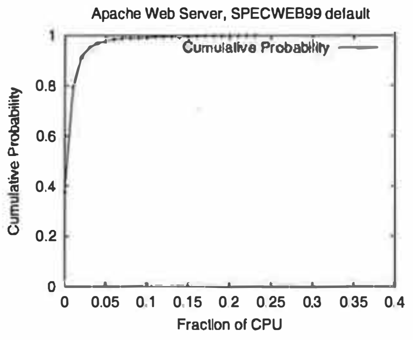
- *PostgreSQL database server:* We profile the postgreSQL database server (version 7.2.1) using the *pgbench 1.2* benchmark. This benchmark is part of the postgreSQL distribution and emulates the TPC-B transactional benchmark [16]. The benchmark provides control over the number of concurrent clients as well as the number of transactions performed by each client. We vary both parameters and study their impact on the resource usage of the database server.

We now present some results from our profiling study.

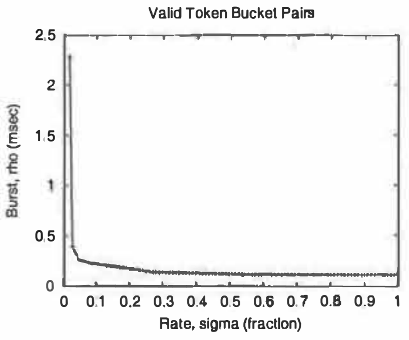
Figure 4(a) depicts the CPU usage distribution of the Apache web server obtained using the default settings of the SPECWeb99 benchmark (50 concurrent clients, 30% dynamic cgi-bin requests). Figure 4(b) plots the corresponding cumulative distribution function (CDF) of the resource usage. As shown in the figure (and summarized in Table 1), the worst case CPU usage (100^{th} percentile) is 25% of CPU capacity. Further, the 99^{th} and the 95^{th} percentiles of CPU usage are 10 and 4% of capacity, respectively. These results indicate that CPU usage is bursty in nature and that the worst-case requirements are significantly higher than a high percentile of the usage. Consequently, under provisioning (i.e., overbooking) by a mere 1% reduces the CPU requirements of Apache by a factor of 2.5, while overbooking by 5% yields a factor of 6.25 reduction (implying that 2.5 and 6.25 times as many web servers can be supported when provisioning based on the 99^{th} and 95^{th} percentiles, respectively, instead of the 100^{th} profile). Thus, even small amounts of overbooking can potentially yield significant increases in platform capacity. Figure 4(c) depicts the possible valid (σ, ρ) pairs for Apache's CPU usage. Depending on the specified overbooking tolerance O, we can set σ to an appropriate



(a) Probability distribution (PDF)



(b) Cumulative distribution function (CDF)



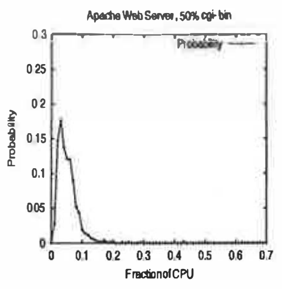
(c) Token bucket parameters

Figure 4: Profile of the Apache web server using the default SPECWeb99 configuration.

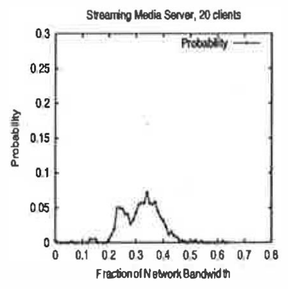
percentile of the usage distribution U, and the corresponding ρ can then be chosen using this figure.

Figures 5(a)-(d) depict the CPU or network bandwidth distributions, as appropriate, for various server applications. Specifically, the figure shows the usage distribution for the Apache web server with 50% dynamic SPECWeb requests, the streaming media server with 20 concurrent clients, the Quake game server with 4 clients and the postgreSQL server with 10 clients. Table 1 summarizes our results and also presents profiles for several additional scenarios (only a small subset of the three dozen profiles obtained from our experiments are presented due to space constraints). Table 1 also lists the worst-case resource needs as well as the 99th and the 95th percentile of the resource usage.

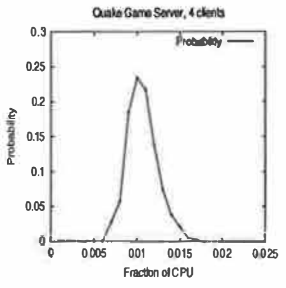
Together, Figure 5 and Table 1 demonstrate that all server applications exhibit burstiness in their resource usage, albeit to different degrees. This burstiness causes the worst-case resource needs to be significantly higher than a high percentile of the usage distribution. Consequently, we find that the 99th percentile is smaller by a factor of 1.1-2.5, while the 95th percentile yields a factor of 1.3-6.25 reduction when compared to the 100th percentile. Together, these results illustrate the potential gains that can be realized by overbooking resources in shared hosting platforms.



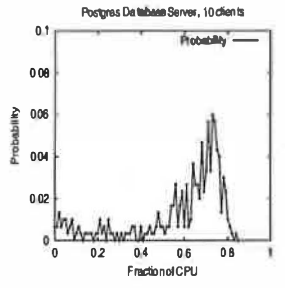
(a) Apache: dynamic requests



(b) Streaming media server



(c) Quake Server



(d) PostgreSQL Server

Figure 5: Profiles of Various Server Applications

3 Resource Overbooking and Capsule Placement in Hosting Platforms

Having derived the QoS requirements of each capsule, the next step is to determine which platform node will run each capsule. Several considerations arise when making such placement decisions. First, since platform resources are being overbooked, the platform should ensure that the QoS requirements of a capsule will be met even in the presence of overbooking. Second, since multiple nodes may have the resources necessary to house each application capsule, the platform will need to pick a specific mapping from the set of feasible mappings. In this section, we present techniques for overbooking platform resources in a controlled manner. The aim is to ensure that: (i) the QoS requirements of the application are satisfied and (ii) overbooking tolerances are taken into account while making placement decisions.

3.1 Resource Overbooking Techniques

A platform node can accept a new application capsule so long as the resource requirements of existing capsules are not violated, and sufficient unused resources exist to meet the requirements of the new capsule. However, if the node resources are overbooked, another requirement is added:

Application	Res.	Res. usage at percentile			(σ, ρ)
		100th	99th	95th	for $O = 0.01$
WS,default	CPU	0.25	0.10	0.04	(0.10, 0.218)
WS, 50% dyn.	CPU	0.69	0.29	0.12	(0.29, 0.382)
SMS,k=4	Net	0.19	0.16	0.11	(0.16, 1.89)
SMS,k=20	Net	0.63	0.49	0.43	(0.49, 6.27)
GS,k=2	CPU	0.011	0.010	0.009	(0.010, 0.00099)
GS,k=4	CPU	0.018	0.016	0.014	(0.016, 0.00163)
DBS,k=1 (def)	CPU	0.33	0.27	0.20	(0.27, 0.184)
DBS,k=10	CPU	0.85	0.81	0.79	(0.81, 0.130)

Table 1: Summary of profiles. Although we profiled both CPU and network usage for each application, we only present results for the more constraining resource due to space constraints. Abbreviations: WS=Apache, SMS=streaming media server, GS=Quake game server, DBS=database server, k=number of clients, dyn.=dynamic, Res.=Resource.

the overbooking tolerances of individual capsules already placed on the node should not be exceeded as a result of accepting the new capsule. Verifying these conditions involves two tests:

Test 1: Resource requirements of the new and existing capsules can be met. To verify that a node can meet the requirements of all capsules, we simply sum the requirements of individual capsules and ensure that the aggregate requirements do not exceed node capacity. For each capsule i on the node, the QoS parameters (σ_i, ρ_i) and τ_i require that the capsule be allocated $(\sigma_i \cdot \tau_i + \rho_i)$ resources in each interval of duration τ_i. Further, since the capsule has an overbooking tolerance O_i, in the worst case, the node can allocate only $(\sigma_i \cdot \tau_i + \rho_i) * (1 - O_i)$ resources and yet satisfy the capsule needs (thus, the overbooking tolerance represents the fraction by which the allocation may be reduced if the node saturates due to overbooking). Consequently, even in the worst case scenario, the resource requirements of all capsules can be met so long as the total resource requirements do not exceed the capacity:

$$\sum_{i=1}^{k+1} (\sigma_i \cdot \tau_{min} + \rho_i) \cdot (1 - O_i) \leq C \cdot \tau_{min} \qquad (1)$$

where C denotes the CPU or network interface capacity on the node, k denotes the number of existing capsules on the node, $k+1$ is the new capsule, and $\tau_{min} = \min(\tau_1, \tau_2, \ldots \tau_{k+1})$ is the period τ for the capsule that desires the most stringent guarantees. [1]

Test 2: Overbooking tolerances of all capsules are met. The overbooking tolerance of a capsule is met only if the total amount of overbooking is smaller than its specified tolerance. To compute the aggregate overbooking on a node, we must first compute the total resource usage on the node. Since the usage distributions U_i of individual capsules are known, the total resource on a node is simply the sum of the individual usages. That is, $Y = \sum_{i=1}^{k+1} U_i$, where Y denotes the of aggregate resource usage distribution on the node. Assuming each U_i is independent, the resulting distribution Y can be computed from elementary probability theory.[2] Given the total resource usage distribution Y, the probability that the total demand exceeds the node capacity should be less than the overbooking tolerance for every capsule, that is,

$$Pr(Y > C) \leq O_i \;\; \forall i \qquad (2)$$

where C denotes the CPU or network capacity on the node. Rather than verifying this condition for each individual capsule, it suffices to do so for the least-tolerance capsule. That is,

$$Pr(Y > C) \leq \min(O_1, O_2, \ldots, O_{k+1}) \qquad (3)$$

where $Pr(Y > C) = \sum_{x=C}^{\infty} Y(x)$. Note that Equation 3 enables a platform to provide a probabilistic guarantee that a capsule's QoS requirements will be met at least $(1 - O_{min}) \times 100\%$ of the time.

Equations 1 and 3 can easily handle heterogeneity in nodes by using appropriate C values for the CPU and network capacities on each node.

A new capsule can be placed on a node if Equations 1 and 3 are satisfied for both the CPU and network interface.

3.2 Capsule Placement Algorithms

Consider an application with m capsules that needs to be placed on a shared hosting platform with N nodes. For each of the m capsules, we can determine the set of *feasible* platform nodes. A feasible node is one that can satisfy the capsule's resource requirements (i.e., satisfies Equations 1 and 3 for both the CPU and network requirements). The platform must then pick a feasible node for each capsule such that all m capsules can be placed on the platform, with the constraint that no two capsules can be placed on the same node (since, by definition, two capsules from the same application are not colocated).

The placement of capsules onto nodes subject to the above constraint can be handled as follows. We model the

[1]Note that since the σ_i for capsule i was chosen based on the $(1 - O_i) * 100^{th}$ percentile of the capsule's resource usage distribution, this multiplication with $(1 - O_i)$ may seem like penalizing the capsule twice. However, this is not so because σ_i in combination with the burst ρ_i is an upper envelop of the requirements of capsule i. The multiplication with $(1 - O_i)$ allows us to overbook the resources on a node in a controlled manner.

[2]This is done using the z-transform. The z-transform of a random variable U is the polynomial $Z(U) = a_0 + za_1 + z^2a_2 + \cdots$ where the coefficient of the i^{th} term represents the probability that the random variable equals i (i.e., $U(i)$). If $U_1, U_2, \ldots, U_{k+1}$ are $k+1$ independent random variables, and $Y = \sum_{i=1}^{k+1} U_i$, then $Z(Y) = \prod_{i=1}^{k+1} Z(U_i)$. The distribution of Y can then be computed using a polynomial multiplication of the z-transforms of $U_1, U_2, \cdots, U_{k+1}$.

placement problem using a graph that contains a vertex for each of the m capsules and N nodes. We add an edge between a capsule and a node if that node is a feasible node for the capsule (i.e., has sufficient resources to house the application). The result is a bipartite graph where each edge connects a capsule to a node.

Given such a graph, we use the following algorithm to determine a placement. The algorithm starts with the capsule that is most constrained (i.e., has the least number of edges/feasible nodes) and places it on any one of its feasible nodes. The node and all of its edges are deleted (since no other capsule can be placed on it). The algorithm then picks the next most constrained capsule and repeats the above process until all m capsules are placed onto nodes. It can be shown that such a greedy algorithm will *always find a placement if one exists* (see an extended version of this paper [24] for a formal proof). This property is used as follows — if no node is found to place a capsule, the algorithm terminates declaring that no placement exists for the application. Further, the algorithm is efficient, since capsules can be placed in a single linear scan once they are sorted in the increasing order of out-degree, resulting in an overall complexity of $O(m \cdot \log m)$.

In our description of the placement algorithm above, we left unspecified how a node is chosen for a capsule out of the many possible feasible nodes. The choice of a particular feasible node can have important implications on the total number of applications supported by the platform. Consequently, we consider four policies for making this decision. The first policy is random, where we pick one feasible node randomly. The second policy is best-fit, where we choose the feasible node which has the least unused resources (i.e., constitutes the best fit for the capsule). The third policy is worst-fit, where we place the capsule onto the feasible node with the most unused resources. In general, the unused network and CPU capacities on a node may be different, and similarly, the capsule may request different amounts of CPU and network resources. Consequently, defining the best and worst fits for the capsule must take into account the unused capacities on *both* resources—we currently do so by simply considering the mean unused capacity across the two resources and compare it to the mean requirements across the two resources to determine the "fit". A fourth policy is to place a capsule onto a node that has other capsules with similar overbooking tolerances. Since a node must always meet the requirements of its least tolerant capsule per Equation 3, colocating capsules with similar overbooking tolerances permits the platform provider to maximize the amount of resource overbooking in the platform. For instance, placing a capsule with a tolerance of 0.01 onto a node that has an existing capsule with $O = 0.05$ reduces the maximum permissible overbooking on that node to 1% (since $O_{min} = \min(0.01, 0.05) = 0.01$). On the other hand, placing this less-tolerant capsule on another node may allow future, more tolerant capsules to be placed onto this node, thereby allowing nodes resources to be overbooked to a greater extent. We experimentally compare the effectiveness of these three policies in Section 5.2.

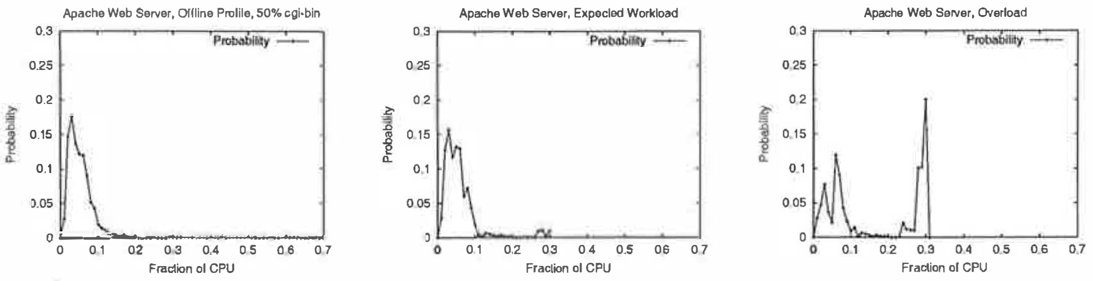
3.3 Handling Dynamically Changing Resource Requirements

Our discussion thus far has assumed that the resource requirements of an application at run-time do not change after the initial profiling phase. In reality though, resource requirements change dynamically over time, in tandem with the workload seen by the application. In this section we outline our approach for dealing with dynamically changing application workloads.

First, recall that we provision resources based on a high percentile of the application's resource usage distribution. Consequently, variations in the application workload that affect only the average resource requirements of the capsules, *but not the tail of the resource usage distribution*, will not result in violations of the probabilistic guarantees provided by the hosting platform. In contrast, workload changes that cause an increase in the tail of the resource usage distribution will certainly affect application QoS guarantees.

How a platform should deal with such changes in resource requirements depends on several factors. Since we are interested in yield management, the platform should increase the resources allocated to an overload application only if it increases revenues for the platform provider. Thus, if an application provider only pays for a fixed amount of resources, there is no economic incentive for the platform provider to increase the resource allocation beyond this limit even if the application is overloaded. In contrast, if the contract between the application and platform provider permits usage-based charging (i.e., charging for resources based on the actual usage, or a high percentile of the usage), then allocating additional resources in response to increased demand is desirable for maximizing revenue. In such a scenario, handling dynamically changing requirements involves two steps: (i) detecting changes in the tail of the resource usage distribution, and (ii) reacting to these changes by varying the actual resources allocated to the application.

To detect such changes in the tail of an application's resource usage distribution, we propose to conduct continuous, on-line profiling of the resource usage of all capsules using low-overhead profiling tools. This would be done by recording the CPU scheduling instants, network transmission times and packet sizes for all processes over intervals of a suitable length. At the end of each interval, this data would be processed to construct the latest resource usage distributions for all capsules. An application



(a) Original profile (b) Profile under expected load (c) Profile under overload

Figure 6: Demonstration of how an application overload may be detected by comparing the latest resource usage profile with the original offline profile.

overload would manifest itself through an increased concentration in the high percentile buckets of the resource usage distributions of its capsules.

We present the results of a simple experiment to illustrate this. Figure 6(a) shows the CPU usage distribution of the Apache web server obtained via offline profiling. The workload for the web server was generated by using the SPECWeb99 benchmark emulating 50 concurrent clients with 50% dynamic cgi-bin requests. The offline profiling was done over a period of 30 minutes. Next, we assumed an overbooking tolerance of 1% for this web server capsule. As described in Section 2.3, it was assigned a CPU rate of 0.29 (corresponding to the 99^{th} percentile of its CPU usage distribution). The remaining capacity was assigned to a greedy *dhrystone* application (this application performs compute-intensive integer computations and greedily consumes all resources allocated to it). The web server was then subjected to exactly the same workload (50 clients with 50% cgi-bin requests) for 25 minutes, followed by a heavier workload consisting of 70 concurrent clients with 70% dynamic cgi-bin requests for 5 minutes. The heavier workload during the last 5 minutes was to simulate an unexpected flash crowd. The web server's CPU usage distribution was recorded over periods of length 10 minute each. Figure 6(b) shows the CPU usage distribution observed for the web server during a period of expected workload. We find that this profile is very similar to the profile obtained using offline measurements, except being upper-bounded by the CPU rate assigned to the capsule. Figure 6(c) plots the CPU usage distribution during the period when the web server was overloaded. We find an increased concentration in the high percentile regions of this distribution compared to the original distribution.

The detection of application overload would trigger remedial actions that would proceed in two stages. First, new resource requirements would be computed for the affected capsules. Next, actions would be taken to provide the capsules the newly computed resource shares — this may involve increasing the resource allocations of the capsules, or moving the capsules to nodes with sufficient resources. Implementing and evaluating these techniques for handling application overloads are part of our ongoing research on shared hosting platforms.

4 Implementation Considerations

In this section, we first discuss implementation issues in integrating our resource overbooking techniques with OS resource allocation mechanisms. We then present an overview of our prototype implementation.

4.1 Providing Application Isolation at Run Time

The techniques described in the previous section allow a platform provider to overbook platform resources and yet provide guarantees that the QoS requirements of applications will be met. The task of enforcing these guarantees at run-time is the responsibility of the OS kernel. To meet these guarantees, we assume that the kernel employs resources allocation mechanisms that support some notion of quality of service. Numerous such mechanisms—such as reservations, shares and token bucket regulators [4, 12, 13]—have been proposed recently. All of these mechanisms allow a certain fraction of each resource (CPU cycles, network interface bandwidth) to be reserved for each application and enforce these allocations on a fine time scale.

In addition to enforcing the QoS requirements of each application, these mechanisms also isolate applications from one another. By limiting the resources consumed by each application to its reserved amount, the mechanisms prevent a malicious or overloaded application from grabbing more than its allocated share of resources, thereby providing application isolation at run-time—an important requirement in shared hosting environments running untrusted applications.

Our overbooking techniques can exploit many commonly used QoS-aware resource allocation mechanisms. Since the QoS requirements of an application are defined in a OS- and mechanism-independent manner, we need to map these OS-independent QoS requirements to mechanism-specific parameter values. We consider three commonly-used QoS-aware mechanisms — reservations, proportional-share schedulers and rate regulators. Due to space constraints, we present only the mapping for reservations here and point the reader to an extended version of this paper [24] for the remaining mappings.

A reservation-based scheduler [12, 13] requires the resource requirements to be specified as a pair (x, y) where the capsule desires x units of the resource every y time units (effectively, the capsule requests $\frac{x}{y}$ fraction of the resource). For reasons of feasibility, the sum of the requests allocations should not exceed 1 (i.e., $\sum_j \frac{x_j}{y_j} \leq 1$). In such a scenario, the QoS requirements of a capsule with token bucket parameters (σ_i, ρ_i) and an overbooking tolerance O_i can be translated to reservation by setting $(1-O_i)\cdot\sigma_i = \frac{x_i}{y_i}$ and $(1-O_i)\cdot\rho_i = x_i$. To see why, recall that $(1 - O_i) \cdot \sigma_i$ denotes the rate of resource consumption of the capsule in the presence of overbooking, which is same as $\frac{x_i}{y_i}$. Further, since the capsule can request x_i units of the resource every y_i time units, and in the worst case, the entire x_i units may be requested continuously, we set the burst size to be $(1 - O_i) \cdot \rho_i = x_i$. These equations simplify to $x_i = (1 - O_i) \cdot \rho_i$ and $y_i = \rho_i/\sigma_i$.

4.2 Prototype Implementation

We have implemented a Linux-based shared hosting platform that incorporates the techniques discussed in the previous sections. Our implementation consists of three key components: (i) a profiling module that allows us to profile applications and empirically derive their QoS requirements, (ii) a control plane that is responsible for resource overbooking and capsule placement, and (iii) a QoS-enhanced Linux kernel that is responsible for enforcing application QoS requirements.

The profiling module runs on a set of dedicated (and therefore isolated) platform nodes and consists of a vanilla Linux kernel enhanced with the Linux trace toolkit. As explained in Section 2, the profiling module gathers a kernel trace of CPU and network activities of each capsule. It then post-processes this information to derive an On-Off trace of resource usage and then derives the usage distribution U and the token bucket parameters for this usage.

The control plane is responsible for placing capsules of newly arriving applications onto nodes while overbooking node resources. The control plane also keeps state consisting of a list of all capsules residing on each node and their QoS requirements. It also maintains information about the hardware characteristics of each node. The requirements of a newly arriving application are specified to the control plane using a resource specification language. This specification includes the CPU and network bandwidth requirements of each capsule. The control plane uses this specification to derive a placement for each capsule as discussed in Section 3.2. In addition to assigning each capsule to a node, the control plane also translates the QoS parameters of the capsules to parameters of commonly used resource allocation mechanisms (discussed in the previous section).

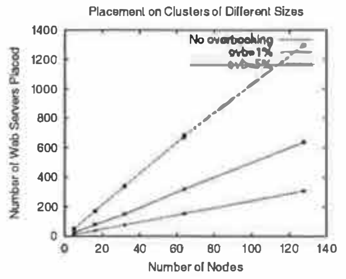
The third component, namely the QoS-enhanced Linux kernel, runs on each platform node and is responsible for enforcing the QoS requirements of capsules at run time. For the purposes of this paper, we implement the H-SFQ proportional-share CPU scheduler [10]. H-SFQ is a *hierarchical* proportional-share scheduler that allows us to group resource principals (processes, lightweight processes) and assign an aggregate CPU share to the entire group. We implement a token bucket regulator to provide QoS guarantees at the network interface card. Our rate regulator allows us to associate all network sockets belonging to a group of processes to a single token bucket. We instantiate a token bucket regulator for each capsule and regulate the network bandwidth usage of all resource principals contained in this capsule using the (σ, ρ) parameters of the capsule's network bandwidth usage. In Section 5.3, we experimentally demonstrate the efficacy of these mechanisms in enforcing the QoS requirements of capsules even in the presence of overbooking.

5 Experimental Evaluation

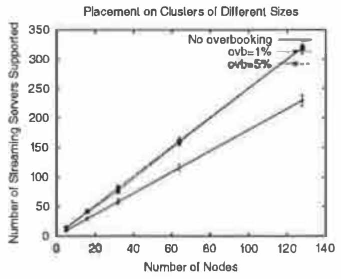
In this section, we present the results of our experimental evaluation. The setup used in our experiments is identical to that described in Section 2.4—we employ a cluster of Linux-based servers as our shared hosting platform. Each server runs a QoS-enhanced Linux kernel consisting of the H-SFQ CPU scheduler and a leaky bucket regulator for the network interface. The control plane for the shared platform implements the resource overbooking and capsule placement strategies discussed earlier in this paper. For ease of comparison, we use the same set of applications discussed in 2.4 and their derived profiles (see Table 1) for our experimental study.

5.1 Efficacy of Resource Overbooking

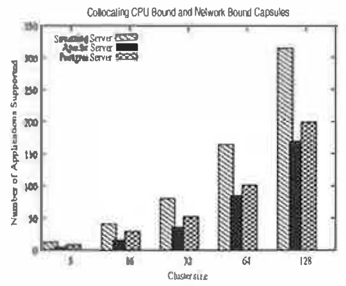
Our first set of experiments examine the efficacy of overbooking resources in shared hosting platforms. We first consider shared *web hosting* platforms — a type of shared hosting platform that runs only web servers. Each web server running on the platform is assumed to conform to one of the four web server profiles gathered from our profiling study (two of these profiles are shown in Table 1; the other two employed varying mixes of static and dynamic



(a) Web servers



(b) Streaming media servers



(c) Mix of applications

Figure 7: Benefits of resource overbooking for a bursty web server application, a less bursty streaming server application and for application mixes.

SPECWeb99 requests). The objective of our experiment is to examine how many such web servers can be supported by a given platform configuration for various overbooking tolerances. We vary the overbooking tolerance from 0% to 10%, and for each tolerance value, attempt to place as many web servers as possible until the platform resources are exhausted. We first perform the experiment for a cluster of 5 nodes (identical to our hardware configuration) and then repeat it for cluster sizes ranging from 16 to 128 nodes (since we lack clusters of these sizes, for these experiments, we only examine how many applications can be accommodated on the platform and do not actually run these applications). Figure 7(a) depicts our results with 95% confidence intervals. The figure shows that, the larger the amount of overbooking, the larger is the number of web servers that can be run on a given platform. Specifically, for a 128 node platform, the number of web servers that can be supported increases from 307 when no overbooking is employed to over 1800 for 10% overbooking (a factor of 5.9 increase). Even for a modest 1% overbooking, we see a factor of 2 increase in the number of web servers that can be supported on platforms of various sizes. Thus, even modest amounts of overbooking can significantly enhance revenues for the platform provider.

Next, we examine the benefits of overbooking resources in a shared hosting platform that runs a mix of streaming servers, database servers and web servers. To demonstrate the impact of burstiness on overbooking, we first focus only on the streaming media server. As shown in Table 1, the streaming server (with 20 clients) exhibits less burstiness than a typical web server, and consequently, we expect smaller gains due to resource overbooking. To quantify these gains, we vary the platform size from 5 to 128 nodes and determine the number of streaming servers that can be supported with 0%, 1% and 5% overbooking. Figure 7(b) plots our results with 95% confidence intervals. As shown, the number of servers that can be supported increases by 30-40% with 1% overbooking when compared to the no overbooking case. Increasing the amount of overbooking from 1% to 5% yields only a marginal additional gain, consistent with the profile for this streaming server shown in Table 1 (and also indicative of the less-tolerant nature of this soft real-time application). Thus, less bursty applications yield smaller gains when overbooking resources.

Although the streaming server does not exhibit significant burstiness, large statistical multiplexing gains can still accrue by colocating bursty and non-bursty applications. Further, since streaming server is heavily network-bound and uses a minimal amount of CPU, additional gains are possible by colocating applications with different bottleneck resources (e.g., CPU-bound and network-bound applications). To examine the validity of this assertion, we conduct an experiment where we attempt to place a mix of streaming, web and database servers—a mix of CPU-bound and network-bound as well as bursty and non-bursty applications. Figure 7(c) plots the number of applications supported by platforms of different sizes with 1% overbooking. As shown, an identical platform configuration is able to support a large number of applications than the scenario where only streaming servers are placed on the platform. Specifically, for a 32 node cluster, the platform supports 36 and 52 additional web and database servers in addition to the approximately 80 streaming servers that were supported earlier. We note that our capsule placement algorithms are automatically able to extract these gains without any specific "tweaking" on our part. Thus, colocating applications with different bottleneck resources and different amounts of burstiness enhance additional statistical multiplexing benefits when overbooking resources.

5.2 Capsule Placement Algorithms

Our next experiment compares the effectiveness of the best-fit, worst-fit and random placement algorithms discussed in Section 3.2. Using our profiles, we construct two types of applications: a replicated web server and an e-commerce application consisting of a front-end web server and a back-end database server. Each arriving application belongs to one of these two categories and is assumed to consist of 2-10 capsules, depending on the degree of replication. The overbooking tolerance is set to 5%. We then determine the number of applications that can be placed on a given platform by different placement strategies. Figure 8(a) depicts our results. As shown, best-fit and random placement yield similar performance, while worst-fit outperforms these two policies across a range of platform sizes. This is because best-fit places capsules onto nodes with smaller unused capacity, resulting in "fragmentation" of unused capacity on a node; the leftover capacity may be wasted if no additional applications can be accommodated. Worst fit, on the other hand, reduces the chances of such fragmentation by placing capsules onto nodes with the larger unused capacity. While such effects become prominent when application capsules have widely varying requirements (as observed in this experiment), they become less noticeable when the application have similar resource requirements. To demonstrate this behavior, we attempted to place Quake game servers onto platforms of various sizes. Observe from Table 1 that the game server profiles exhibit less diversity than a mix of web and database servers. Figure 8(b) shows that, due to the similarity in the application resource requirements, all policies are able to place a comparable number of game servers.

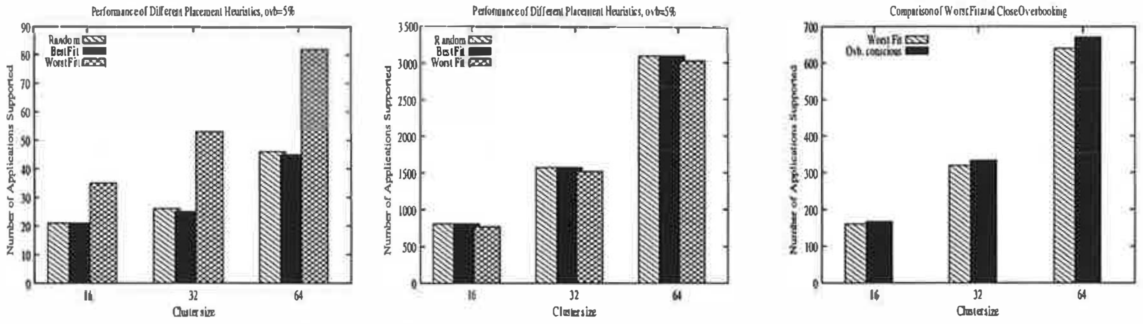
Finally, we examine the effectiveness of taking the overbooking tolerance into account when making placement decisions. We compare the worst-fit policy to an overbooking-conscious worst-fit policy. The latter policy chooses the three worst-fits among all feasible nodes and picks the node that best matches the overbooking tolerance of the capsule. Our experiment assumes a web hosting platform with two types of applications: less-tolerant web servers that permit 1% overbooking and more tolerant web servers that permit 10% overbooking. We vary the platform size and examine the total number of applications placed by the two policies. As shown in Figure 8(c), taking overbooking tolerances into account when making placement decisions can help increase the number of applications placed on the cluster. However, we find that the additional gains are small ($< 6\%$ in all cases), indicating that a simple worst-fit policy may suffice for most scenarios.

5.3 Effectiveness of Kernel Resource Allocation Mechanisms

While our experiments thus far have focused on the impact of overbooking on platform capacity, in our next experiment, we examine the impact of overbooking on application performance. We show that combining our overbooking techniques with kernel-based QoS resource allocation mechanisms can indeed provide application isolation and quantitative performance guarantees to applications (even in the presence of overbooking). We begin by running the Apache web server on a dedicated (isolated) node and examine its performance (by measuring throughput in requests/s) for the default SPECWeb99 workload. We then run the web server on a node running our QoS-enhanced Linux kernel. We first allocate resources based on the 100^{th} percentile of its usage (no overbooking) and assign the remaining capacity to a greedy *dhrystone* application. We measure the throughput of the web server in presence of this background dhrystone application. Next, we reserve resources for the web server based on the 99^{th} and the 95^{th} percentiles, allocate the remaining capacity to the dhrystone application, and measure the server throughput. Table 2 depicts our results. As shown, provisioning based on the 100^{th} percentile yields performance that is comparable to running the application on an dedicated node. Provisioning based on the 99^{th} and 95^{th} percentiles results in a small degradation in throughput, but well within the permissible limits of 1% and 5% degradation, respectively, due to overbooking. Table 2 also shows that provisioning based on the average resource requirements results in a substantial fall in throughout, indicating that reserving resources based on mean usage is not advisable for shared hosting platforms.

We repeat the above experiment for the streaming server and the database server. The background load for the streaming server experiment is generated using a greedy UDP sender that transmits network packets as fast as possible, while that in case of the database server is generated using the dhrystone application. In both cases, we first run the application on an isolated node and then on our QoS-enhanced kernel with provisioning based on the 100^{th}, 99^{th} and the 95^{th} percentiles. We also run the application with provisioning based on the average of its resource usage distribution obtained via offline profiling. We measure the throughput in transaction/s for the database server and the mean length of a playback violation (in seconds) for the streaming media server. Table 2 plots our results. Like with the web server, provisioning based on the 100^{th} percentile yields performance comparable to running the application on an isolated node, while a small amount of overbooking results in a corresponding small amount of degradation in application performance.

For each of the above scenarios, we also computed the application profiles in the presence of background load and overbooking and compared these to the profiles gathered on the isolated node. Figure 9 shows one such set of profiles. It should be seen in combination with the second row in Table 2 that corresponds to the PostgreSQL application. Together, they depict the performance of the



(a) Placing diverse applications (b) Placing similar applications (c) Overbooking-conscious placement

Figure 8: Performance of various capsule placement strategies.

Application	Metric	Isolated Node	100th	99th	95th	Average
Apache	Throughput (req/s)	67.93 ± 2.08	67.51 ± 2.12	66.91 ± 2.76	64.81 ± 2.54	39.82 ± 5.26
PostgreSQL	Throughput (transactions/s)	22.84 ± 0.54	22.46 ± 0.46	22.21 ± 0.63	21.78 ± 0.51	9.04 ± 0.85
Streaming	Length of violations (sec)	0	0	0.31 ± 0.04	0.59 ± 0.05	5.23 ± 0.22

Table 2: Effectiveness of kernel resource allocation mechanisms. All results are shown with 95% confidence intervals.

database server for different levels of CPU provisioning. Figures 9(b) and (c) show the CPU profiles of the database server when it is provisioned based on the 99th and the 95th percentiles respectively. As can be seen, the two profiles look similar to the original profile shown in Figure 9(a). Correspondingly, Table 2 shows that for these levels of CPU provisioning, the throughput received by the database server is only slightly inferior to that on an isolated node. This indicates that upon provisioning resources based on a high percentile, the presence of background load interferes minimally with the application behavior. In Figure 9(d), we show the CPU profile when the database server was provisioned based on its average CPU requirement. This profile is drastically different from the original profile. We also present the corresponding low throughput in Table 2. This reinforces our earlier observation that provisioning resources based on the average requirements can result in significantly degraded performance.

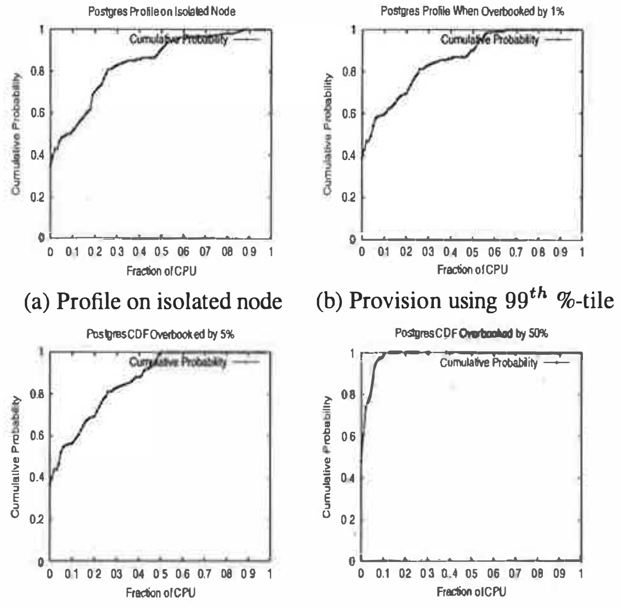
Together, these results demonstrate that our kernel resource allocation mechanisms are able to provide quantitative performance guarantees even when resources are overbooked.

6 Related Work

Research on clustered environments over the past decade has spanned a number of issues. Systems such as Condor have investigated techniques for harvesting idle CPU cycles on a cluster of workstations to run batch jobs [15]. The design of scalable, fault-tolerant network services running on server clusters has been studied in [8]. Use of virtual clusters to manage resources and contain faults in large multiprocessor systems has been studied in [9]. Scalability, availability and performance issues in dedicated clusters have been studied in the context of clustered mail servers [18] and replicated web servers [3]. Ongoing efforts in the grid computing community have focused on developing standard interfaces for resource reservations in clustered environments [11]. In the context of QoS-aware resource allocation, numerous efforts over the past decade have developed predictable resource allocation mechanisms for single machine environments [4, 12, 13].

Statistical admission control techniques that overbook resources have been studied in the context of video-on-demand servers [25] and ATM networks [6], but little work as been published to date in the context of shared, cluster-based hosting platforms.

Most closely related to our work is Aron [1, 2], who presents a comprehensive framework for resource management in web servers, with the aim of delivering predictable QoS and differentiated services. New services are profiled by running on lightly-loaded machines, and contracts subsequently negotiated in terms of application level performance (connections per second), reported by the application to the system. CPU and disk bandwidth are scheduled by lottery scheduling [26] and SFQ [10] respectively, while physical memory is statically partitioned between services with free pages allocated temporarily to services that can make use of them. A *resource monitor* running over a longer timescale examines performance



(a) Profile on isolated node (b) Provision using 99th %-tile

(c) Provision using 95th %-tile (d) Provision using average

Figure 9: Effect of different levels of provisioning on the PostgreSQL server CPU profile.

reported by the application and system performance information and flags conditions which might violate contracts, to allow extra resources to be provided by external means.

In Aron's system, resource allocation is primarily driven by application feedback and the primary concern is allowing a principal to meet its contract. It is instructive to compare this with our own goal of maximizing the *yield* in the system, which amounts to maximizing the proportion of system resources used to satisfy contracts. Over a long time period (over which new machines can be provisioned, for instance) these goals coincide, but as [19] makes clear, over short periods of time they do not. This difference corresponds to a different kind of relationship between service provider and platform provider. Consequently, Aron's system is able to take advantage of uniform application-reported performance metrics; ours in contrast is oriented toward a heterogeneous mixture of services which are untrusted by the platform and potentially antagonistic.

This subtle but important difference also motivates other differences in design choices between the two systems. In Aron's work, application overload is detected when resource usage exceeds some predetermined threshold. We, on the other hand, detect overload by observing the tail of the recent resource usage distributions. On the other hand, several features of Aron's work, such as the use of multiple random variables to capture the behavior of services over different time scales, are directly applicable to our system.

The specific problem of QoS-aware resource management for clustered environments has been investigated in [3]. This effort builds upon single node QoS-aware resource allocation mechanisms and propose techniques to extend their benefits to clustered environments. [7] proposes a system called Muse for provisioning resources in hosting centers based on energy considerations. Muse is based on an economic approach to managing shared server resources in which services "bid" for resources as a function of delivered performance. It also provides mechanisms to continuously monitor load and compute new resources by estimating the value of their effects on service performance. economic approach for sharing resources in such driven by energy considerations. A salient difference between Muse and our approach is that Muse provisions resources based on the *average* resource requirements whereas we provision based on the *tail* of the resource requirements.

In [27], the authors consider a model of hosting platforms different from that considered in our work. They visualize future applications executing on platforms constructed by clustering multiple, autonomous distributed servers, with resource access governed by agreements between the owners and the users of these servers. They present an architecture for distributed, coordinated enforcement of resource sharing agreements based on an application-independent way to represent resources and agreements. In this work we have looked at hosting platforms consisting of servers in one location and connected by a fast network. However we also believe that distributed hosting platforms will become more popular and resource management in such systems will pose several challenging research problems.

7 Concluding Remarks

In this paper, we presented techniques for provisioning CPU and network resources in shared hosting platforms running potentially antagonistic third-party applications. We argued that provisioning resources solely based on the worst-case needs of applications results in low average utilization, while provisioning based on a high percentile of the application needs can yield statistical multiplexing gains that significantly increase the utilization of the cluster. Since an accurate estimate of an application's resource needs is necessary when provisioning resources, we presented techniques to profile applications on dedicated nodes, possibly while in service, and used these profiles to guide the placement of application components onto shared nodes. We then proposed techniques to overbook cluster resources in a controlled fashion such that the platform can provide performance guarantees to applications even when overbooked. Our techniques, in conjunction with commonly used OS resource allocation mechanisms, can provide application isolation and performance guarantees at run-time in the presence of overbooking. We implemented our techniques in a Linux cluster and evaluated them using common server applications. We found that the efficiency benefits from controlled overbooking of resources can be dramatic when compared to provisioning resources based on the worst-case requirements of applications. Specifically, overbooking resources by as little as 1% increases the utilization of the hosting platform by a factor of 2, while overbooking by 5-10% results in gains of up to 500%. The more bursty the application resources needs, the higher are the benefits of resource overbooking. More generally, our results demonstrate the benefits and feasibility of overbooking resources for the platform provider.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Peter Druschel for their comments.

References

[1] M. Aron. Differentiated and Predictable Quality of Service in Web Server Systems. *PhD Thesis, Computer Science, Rice University*, October 2000.

[2] M. Aron, S. Iyer, and P. Druschel. A Resource Management Framework for Predictable Quality of Service in Web Servers. Submitted for publication

[3] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the ACM SIGMETRICS Conference, Santa Clara, CA*, June 2000.

[4] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the third Symposium on Operating System Design and Implementation (OSDI'99), New Orleans*, pages 45–58, February 1999.

[5] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI'02), Boston, MA*, December 2002.

[6] R. Boorstyn, A. Burchard, J. Liebeherr, and C. Oottamakorn. Statistical Service Assurances for Traffic Scheduling Algorithms. *IEEE Journal on Selected Areas in Communications*, 18(12):2651–2664, December 2000.

[7] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 103–116, October 2001.

[8] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based Scalable Network Services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP'97), Saint-Malo, France*, pages 78–91, December 1997.

[9] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource Management using Virtual Clusters on Shared-memory Multiprocessors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99), Kiawah Island Resort, SC*, pages 154–169, December 1999.

[10] P. Goyal, X. Guo, and H.M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of Operating System Design and Implementation (OSDI'96), Seattle*, pages 107–122, October 1996.

[11] Global Grid Forum: Scheduling and Resource Management Working Group. http://www-unix.mcs.anl.gov/ schopf/ggf-sched, 2002.

[12] M B. Jones, D Rosu, and M Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the sixteenth ACM symposium on Operating Systems Principles (SOSP'97), Saint-Malo, France*, pages 198–211, December 1997.

[13] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communication*, 14(7):1280–1297, September 1996.

[14] Linux Trace Toolkit Project Page. http://www.opersys.com/LTT/, 2002.

[15] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[16] *The pgbench man page, PostgreSQL software distribution*, 2002.

[17] T. Roscoe and B. Lyles. Distributing Computing without DPEs: Design Considerations for Public Computing Platforms. In *Proceedings of the 9th ACM SIGOPS European Workshop, Kolding, Denmark*, September 2000.

[18] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: A Highly Available, Scalable Cluster-based Mail Service. In *Proceedings of the 17th SOSP, Kiawah Island Resort, SC*, pages 1–15, December 1999.

[19] B C. Smith, J F. Leimkuhler, and R M. Darrow. Yield Management at American Airlines. *Interfaces*, 22(1):8–31, January-February 1992.

[20] The Standard Performance Evaluation Corporation (SPEC), http://www.spec.org. *SPECWeb99 Benchmark Documentation.*

[21] V Sundaram, A. Chandra, P. Goyal, P. Shenoy, J Sahni, and H Vin. Application Performance in the QLinux Multimedia Operating System. In *Proceedings of the Eighth ACM Conference on Multimedia, Los Angeles, CA*, November 2000.

[22] P. Tang and T. Tai. Network Traffic Characterization Using Token Bucket Model. In *Proceedings of IEEE Infocom'99, New York, NY*, March 1999.

[23] B. Urgaonkar and P. Shenoy. Sharc: Managing CPU and Network Bandwidth in Shared Clusters. Technical Report TR01-08, Department of Computer Science, University of Massachusetts, October 2001.

[24] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. Technical report TR02-21, Department of Computer Science, University of Massachusetts, May 2002.

[25] H. M. Vin, P. Goyal, A. Goyal, and A. Goyal. A Statistical Admission Control Algorithm for Multimedia Servers. In *Proceedings of the ACM Multimedia'94, San Francisco*, pages 33–40, October 1994.

[26] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-share Resource Management. In *Proceedings of symposim on Operating System Design and Implementation*, November 1994.

[27] T. Zhao and V. Karmacheti. Enforcing Resource Sharing Agreements among Distributed Server Clusters. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002.

An Integrated Experimental Environment for Distributed Systems and Networks

Brian White Jay Lepreau Leigh Stoller Robert Ricci Shashi Guruprasad
Mac Newbold Mike Hibler Chad Barb Abhijeet Joglekar

School of Computing, University of Utah

`www.flux.utah.edu` `www.netbed.org`

Abstract

Three experimental environments traditionally support network and distributed systems research: network emulators, network simulators, and live networks. The continued use of multiple approaches highlights both the value and inadequacy of each. Netbed, a descendant of Emulab, provides an experimentation facility that integrates these approaches, allowing researchers to configure and access networks composed of emulated, simulated, and wide-area nodes and links. Netbed's primary goals are *ease of use*, *control*, and *realism*, achieved through consistent use of virtualization and abstraction.

By providing operating system-like services, such as resource allocation and scheduling, and by virtualizing heterogeneous resources, Netbed acts as a virtual machine for network experimentation. This paper presents Netbed's overall design and implementation and demonstrates its ability to improve experimental automation and efficiency. These, in turn, lead to new methods of experimentation, including automated parameter-space studies within emulation and straightforward comparisons of simulated, emulated, and wide-area scenarios.

1 Introduction

The diverse requirements of network and distributed systems research are not well met by any single experimental environment. Competing approaches remain popular because each covers a different point in a space defined by levels of *ease of use*, *control*, and *realism*. Packet-level discrete event simulation and live network experimentation represent two extremes. Simulation presents a controlled, repeatable environment. However, its level of abstraction may be too high to capture low-level effects such as the impact of interrupts under heavy load. Live networks achieve realism, but surrender repeatability and the ability to modify or even monitor internal router behavior. Emulation [1, 27, 36, 42] is a hybrid approach that subjects real applications, protocols, and operating systems to a synthetic network environment. While single-node WAN emulators, such as Dummynet [36], introduce artificial delays, losses, and bandwidth constraints in a controlled manner, they require tedious manual configuration.

Netbed complements existing experimental environments. It spans simulation, emulation, and live network experimentation by integrating them into a common framework. This integration brings the control and ease of use usually associated with simulation to emulation and live network experimentation without sacrificing realism. It gives users the individual benefits of simulation, emulation, and live network experimentation, configured and controlled in a consistent manner. Further, integration facilitates interaction, comparison, and validation across the three domains.

Netbed is a software system that provides a time- and space-shared platform for research, education, or development in distributed systems and networks. It leverages local nodes, allocated from clusters and temporarily dedicated to individual users, for emulation; this paper often refers to these as emulated nodes. Netbed also employs geographically-distributed nodes that are simultaneously shared amongst users; this paper frequently refers to such resources as wide-area nodes. Researchers access these resources by specifying a virtual topology graphically or via an *ns* script [40], causing Netbed to automatically configure a physical topology. An *experiment* is defined by this configuration and any run-time dynamics, such as traffic generation, specified via the general-purpose *ns* interface. When realizing the virtual topology, Netbed virtualizes host names, IP addresses, links, and nodes. Virtual nodes may be instantiated from a large set of local nodes, from a smaller set of distributed nodes, or within *ns* simulation. Virtual links may map directly to local-area links, may be matched to similar wide-area links, or may be emulated by interposing Dummynet nodes to regulate bandwidth, latency, loss, and queuing behavior.

Netbed's framework provides integrated abstractions, services, and name spaces common to all three environments, mapping them into domain-specific mechanisms and internal names. Netbed's operating system-like services include node and link allocation and naming, scheduling and idle experiment preemption, experiment "swapping," and disk image loading.

Given these services, an analogy between an experiment and a Unix process seems natural. This metaphor illustrates the life cycle of an experiment and Netbed's role in automating and controlling the procedure. The *ns* specification serves as the "program text," which Netbed compiles to synthesize a hardware realization of the virtual topology. The specification is first parsed into an intermediate representation that is stored in a database and later allocated and loaded onto hardware. During experiment execution, Netbed provides interfaces and tools for experiment control and interaction. Finally, Netbed may preempt and swap out an experiment. Because Netbed gives experimenters run-time control over node and link characteristics and an ability to interpose traffic-shaping and monitoring nodes, we view the system as a virtual machine for heterogeneous node, link, and topology allocation and control. While traditional virtual machines target an architecture's instruction set, Netbed instead abstracts the network.

The analogy is not merely cosmetic; experiments derive key benefits from Netbed's design, namely automation and time- and space-efficiency. Experiment creation involves a large number of steps including, for example, configuring network interfaces and routing tables, installing operating systems, exporting file trees, and administering user accounts. Netbed removes the tedium of manual configuration through automation. Netbed was designed to make efficient use of physical resources and to enhance experimenter productivity. It manages the shared use of physical resources to provide their greatest possible utilization, while ensuring inter-experiment isolation. Netbed performs experiment creation and termination in a few minutes, enabling an interactive style of use. Attention to efficiency of disk reloading, resource allocation, and experiment creation maximizes time spent executing experiments and minimizes effort expended configuring them.

This paper makes the following contributions:

- It introduces the notion of a virtual machine for controlled network experimentation and shows how it integrates heterogeneous resources.
- It outlines the key obstacles to virtual machine efficiency and how they were overcome.
- It shows that Netbed's automation, efficiency, and services inspire qualitatively new methods of experimentation.
- It provides data validating Netbed's emulation capabilities.

Section 2 continues by outlining the heterogeneous resources managed by Netbed. Section 3 outlines the life cycle of an experiment, using the virtual machine analogy to describe the system's design, and Section 4 shows the benefits of this approach. Section 5 details the challenges overcome by Netbed's experiment services, including the mapping of virtual to physical resources and disk loading, and their efficiency. Section 6 validates the emulation facilities. Section 7 illustrates unique experimental techniques facilitated by Netbed. Finally, related work is addressed in Section 8 and Section 9 concludes.

2 Resources

As its original name, "Emulab," suggests, Netbed was conceived as an emulation platform. Through its flexible design, it has evolved to support a diverse set of physical node and link types. These resources are virtualized in the sense that they may be allocated and controlled largely independently of their physical realization.

Local-Area Resources: Netbed software currently controls two clusters: one at the University of Utah comprised of 168 PCs and another at the University of Kentucky containing 50 PCs. The two sites are configured in a nearly identical fashion. Any of these nodes can function as an edge node, a traffic generator, or a router. Each machine has five 100Mb Ethernet interfaces: one is on a dedicated control and data acquisition network and the others are for arbitrary use by experiments. At each node, local memory and disk provide ample room for computation and logging of monitoring data.

All local nodes are connected using high-end switches that function as a "programmable patch panel." To support arbitrary and isolated topologies and to provide security to Netbed users, we employ Virtual LANs. A VLAN is a switch technology that restricts traffic to the subnet defined by its members. We have verified empirically that our switches provide inter-VLAN performance isolation, in the face of both traffic and control operations (VLAN creation, deletion, and modification).

Netbed's local nodes and wealth of available bandwidth can be configured into switched LAN topologies. This, coupled with its rapid and automated configuration of operating systems, makes Netbed an attractive platform for kernel development and research within local-area networks. Root privileges, remotely accessible consoles, and remote power cycling help make kernel development convenient.

Emulated Resources: Netbed uses Dummynet and VLANs to emulate wide-area links within the local-area environment. A Dummynet node is automatically inserted between two physical nodes and enforces queue and bandwidth limitations, introducing delays and packet loss. Dummynet nodes act as Ethernet bridges and are transparent to experimental traffic.

Distributed Resources: Netbed integrates both the

MIT-owned testbed nodes first used for the RON [4] research, as well as nodes contributed by other organizations that run our special CD-based Unix configuration. These resources today provide Netbed with approximately 40 nodes at 30 different sites around the world, including nodes connected via Internet2, DSL, and cable modems. These nodes are valuable to experimenters performing Internet measurement or who require the characteristics of a live network. Experimenters may request a random set of nodes, specific nodes, nodes having a specific class of network connection (e.g., via a cable modem), or nodes connected via specified latencies, bandwidths, and loss rates. In the latter case, Netbed provides a best-effort mapping of a user-specified virtual topology onto physical distributed nodes.

Distributed nodes support many of Netbed's key features, including account establishment and automated traffic generation, subject to their particular policies and mechanisms. For example, distributed nodes typically have only one network interface, so do not have a physically separate control network. Due to their scarcity, by policy—not limitation of mechanism—distributed nodes currently are shared; multiple experiments may use a node simultaneously. Netbed provides some isolation between experiments through the FreeBSD Jail [18] mechanism, which provides a primitive form of virtual machine and restricts root privileges. Our modifications to Jail provide access to raw sockets, while preventing processes from spoofing IP addresses. Multiplexing is supported by providing a (currently fixed) number of jailed virtual machines per node. Extending this mechanism to provide fair sharing of CPU, memory, and network resources is a subject of future work.

Netbed provides flexibility in specifying interconnections between distributed nodes. By default, the nodes retain full, unmediated access to the Internet. However, if links are specified between the nodes, Netbed sets up IP tunnels so that distributed nodes can use "private" IP addresses. In conjunction with Netbed's automated routing setup, this creates an overlay network configured to the experimenter's specifications. These tunnels also allow transparent communication between distributed nodes and experimental interfaces on local nodes, so that networks can contain both Internet and emulated links. Thus, distributed nodes may be treated the same as local nodes with respect to traffic generation, routes, and IP addresses.

Simulated Resources: Netbed integrates simulation through *ns*'s emulation facility, *nse* [10], allowing simulated nodes, links, and traffic to interact with application traffic. Though simulation abstracts detail [15, 11], it can provide scalability beyond the limits of physical resources; many virtual simulated nodes can be multiplexed on one physical node.

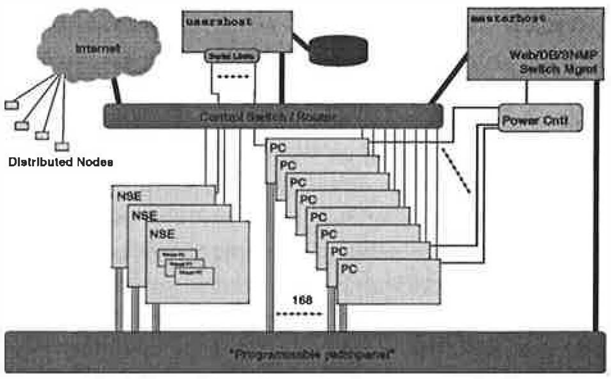


Figure 1: Netbed Architecture

Netbed's deployment of *ns* brings a wealth of simulation infrastructure to emulated and distributed experiments, including *ns*'s rich and diverse protocol suite, varied statistical models, and support for wireless devices. *nse* can also be used to simulate a large-scale network within emulation. The close interaction between simulation and live protocols presents an opportunity to validate *ns*'s abstractions.

Planned Extensions: Plans are underway to integrate additional virtual resource types: we are constructing a WAN emulator based on the Intel IXP1200 network processor [17] that provides improved features and performance over Dummynet. Second, we plan to control and configure ModelNet [42] through Netbed's existing interfaces.

3 Experiment Life Cycle

An experiment is Netbed's central operational entity. It represents a network configuration, including links and VLANs; node state, including operating system images; and database entries, including event sequences. The intended duration of an experiment ranges from a few minutes, to many days, to months or years on distributed nodes. This section follows the life cycle of an experiment to illustrate Netbed's operation and further develop its role as a virtual machine for network experimentation.

The Netbed virtual machine is architected around interacting state machines, monitored by a state management daemon. A primary state machine represents the experiment, while subsidiary state machines handle node allocation, configuration, and disk reloading. The state daemon catches illegal or tardy state transitions. For example, if a node hangs while rebooting, the state daemon times out and attempts an alternate reboot mechanism. This approach copes reasonably well with the reliability challenges of large-scale distributed systems which are composed of often unstable commodity hardware, but further work on reliability is needed.

3.1 Accessing Netbed

To minimize administrative overhead, Netbed employs a hierarchical structure for authorization: To begin a new *project*, a "leader," e. g., a faculty member or senior student, submits a simple web form. Once the project has been approved by Netbed staff, accountability and ability to authorize other project members are delegated to the project leader. The web interface then serves as a universally-accessible portal to Netbed, through which an experimenter may create or terminate an experiment, view the corresponding virtual topology, or configure node properties.

After experiment creation, experimenters may log directly into their allocated nodes, or in to `usershost`, depicted in Figure 1, which serves as a centralized point of control. This node is also `fileserver`, which stores operating system images, exports home and project directories to local nodes via NFS and to distributed nodes via SFS, the Secure File System [20]. `masterhost` is a secure server for many of our critical systems, including the web server, database, and switch management.

3.2 Specification

Just as program text is the concrete specification of a run-time process, an *ns* script written in Tcl configures a Netbed experiment. This choice facilitates validation and comparison since *ns*-specified topologies, traffic generation, and events can be reproduced in an emulated or wide-area environment. For the large community of researchers familiar with *ns*, it provides a graceful transition from simulation and an opportunity to leverage existing scripts. Since Tcl is a general-purpose programming language, a researcher is empowered with looping constructs, conditionals, and arbitrary functions to drive experiment configuration and execution.

Emulated nodes and links enjoy full implementation transparency. By default, links specified in the *ns* experiment file are realized as interposed Dummynet nodes. To instead incorporate distributed nodes, an experimenter need only specify an appropriate node type. For example, Figure 2 requests an Internet-connected node by specifying a `pc-inet` hardware type. A simulated topology can be embedded within an emulated topology by wrapping standard *ns* syntax in a `make-simulated` block, a Netbed-specific construct.

Any constant bit rate traffic flow identified via standard *ns* syntax automatically instantiates traffic sources and sinks using the TG Tool Set [21]. Simulated FTP and Telnet flows are rendered using *ns*'s emulation facility, *nse*. This mechanism injects traffic generated by models, such as the tcplib telnet distribution, into a live network. Such cross traffic is important for studying protocol behavior in the face of congestion.

Netbed defines a small set of *ns* extensions, including procedures to configure a node's operating system and to specify its hardware type. These procedures are not required; Netbed supplies default behavior in their absence. A stub library defines null procedures so that the same script may be executed on Netbed and within *ns*.

```
set ns [new Simulator] # Create the simulator
source tb_compat.tcl   # Add Netbed commands
$ns rtproto Static     # Netbed computes routes

set source [$ns node]  # define new nodes
set router [$ns node]
set dest   [$ns node]

# Connect source to router and router to dest
$ns duplex-link $source $router 10Mb 0ms RED
$ns duplex-link $router $dest 1.5Mb 20ms DropTail

tb-set-node-os  $source FBSD45-STD # Set OS on local node
tb-set-hardware $dest pc-inet   # Request distributed node

$ns run                         # "run" on Netbed
```

Figure 2: An *ns* file showing a linear topology with routing and a distributed node

Program objects are a Netbed-specific *ns* extension that provides a rudimentary remote execution facility. A program object is associated with an *ns* node in the script and attaches arbitrary applications to the corresponding local node. It may be independently controlled during an experiment's execution. Program objects are currently not available on distributed nodes, until we finish securing the distributed event system.

Experimenters unfamiliar with *ns* syntax may create topologies graphically via a Java GUI, which generates an *ns* configuration file. Alternatively, a standard topology generator such as GT-ITM or BRITE may be used to generate an *ns* script. This highlights one of the primary benefits of integration: application of tools intended for one experimental domain, in this case simulation, to another.

3.3 Parsing

A traditional compiler is separated into front and back ends whose interactions are mediated by an intermediate representation. This aids portability since the same front end can be reused with back ends supporting different hardware architectures. Since Netbed targets multiple, heterogeneous physical resources simultaneously, it uses an analogous split-phase style of compilation. A database serves as the shared repository between the front-end Tcl/*ns* parser and resource-specific back-end mechanisms. Thus, a single experiment may incorporate simulated, emulated, and wide-area links without requiring excessive resource-specific knowledge in the specification language or front-end parser.

Netbed's parser recognizes the subset of *ns* relevant to topology and traffic generation. Written in Tcl, it operates by overriding and interposing on standard *ns* procedures and Tcl primitives. Netbed executes the experiment configuration script in the context of these new definitions. Unrecognized *ns* commands output a warning, while *ns* syntax configuring links and traffic endpoints triggers the overloaded procedures. *ns*-specified event generation is performed at this time, storing the events in the database. Therefore, *ns*-specified events are static and have a (large) limit on their number.

Both overloaded and Netbed-specific procedures populate the database, which also stores information about hardware, users, and experiments. The database presents a consistent abstraction of heterogeneous resources to higher layers of Netbed and to experimenters. For example, the front-end database representations of distributed and emulated nodes differ only in a type tag. The database provides a single name space for all experimental entities. Thus, in most cases, experimenters can interact with them using the same commands, tools, and naming conventions regardless of their implementation. As an example, nodes of any type may host traffic generators, despite the fact that the traffic may flow over links simulated by *ns*, emulated by Dummynet, or across the Internet between distributed nodes.

3.4 Global Resource Allocation

The global resource allocation phase is responsible for binding abstractions created during previous stages to physical entities. It corresponds to the resource allocation performed during back-end compilation and linker-directed name binding. For overall simplicity, resources are currently allocated on demand rather than reserved by experimenters in advance.

Netbed uses general combinatorial optimization techniques to perform resource allocation. The algorithms map a target configuration, stored in the database, onto available physical resources. Such a mapping respects the interconnections of the virtual topology, including their latency, bandwidth, and loss rates. As further explained in Sections 5.2 and 5.3, we use separate algorithms for local and distributed nodes due to their differing constraints. The mapping program for local nodes, `assign`, uses simulated annealing, while the `wanassign` program uses a genetic algorithm for distributed resources. Based on the output of `assign` and `wanassign`, Netbed reserves nodes and links and updates the database with resource mappings and user-supplied parameters.

Although within an experiment we follow our principle of conservative resource allocation, we've found it impractical to do so between experiments on local nodes. We currently have only 2 Gbps inter-switch bandwidth, much of which is theoretically consumed by single experiments, preventing other experiments from mapping successfully. However, our traffic monitoring has shown that, in practice, experiments rarely use their allocated inter-switch bandwidth. Therefore we have adopted a policy of over-reserving these bottleneck links while continuously monitoring them for high bandwidth use. Thus far, that has never occurred.

Occasionally, there is a need to dynamically change node membership in an experiment. This can happen, for example, if a node fails and must be replaced, or if nodes are no longer needed because of a change in application demands. Netbed supports the dynamic addition or removal of nodes in any active experiment, and can graft added nodes into LAN-connected topologies.

To ensure consistent naming across instantiations of an *ns* configuration, Netbed virtualizes IP addresses and host names. This level of indirection is necessary since a configuration is unlikely to be mapped to the same physical resources upon re-creation. While experimenters are free to manually assign IP addresses, this task is most often left to Netbed. Netbed deterministically names nodes and links for consistency across experiment creations.

3.5 Node Self-Configuration

Node configuration is driven by the nodes themselves, but entirely controlled by state stored centrally in the database. This is accomplished in a manner reminiscent of Unix dynamic linking and loading. A traditional dynamic linker is responsible for establishing the proper context for a process, loading it, and then invoking it. Netbed applies this strategy at the node level to achieve distributed self-configuration, which includes obtaining a host name, loading a disk image, and executing experiment startup scripts.

Intelligent *node state* management is crucial in realizing our robustness and security goals. Nodes are kept free of persistent configuration state; their memory and local disks are considered volatile soft state. This allows an experiment to be "swapped out" and its resources reclaimed. If experimenters wish to retain local disk modifications, such as kernel revisions, they can easily save an image of their disk on persistent store. A reference to the image is stored in the database and becomes hard state. While an experiment is swapped out, Netbed stores its virtual topology, host name, and general setup in the database. "Swap in" reconstitutes this hard state on an equivalent set of physical resources and brings the node to a fully-known state.

For local nodes, Netbed ensures that a clean disk image is installed on every node before experiment swap-in or creation. Then, in parallel, Netbed attempts to reboot all the nodes using increasingly aggressive techniques. First, it issues a `reboot` command via `ssh`; any nodes that fail to boot in a timely manner are sent a secure authenticated "ping of death"; should that fail, they are power-cycled. Nodes boot using Intel's PXE [34] network bootstrap protocol. Each node's PXE BIOS contacts `masterhost`, which loads a first level kernel as directed by the database. This first level kernel might be a fast disk image loader, a memory file system-based operating system, or typically, a larger second level bootstrap program. This second level loader again contacts the database to determine the next step, either booting from an on-disk partition or downloading an OSKit [12] kernel. This multi-phase approach permits flexible configuration and customization of the OS that runs on each node. The system then waits for the nodes to come back up. If a node does not come up in a timely manner, one more attempt is made; if it still fails, the entire experiment swap-in fails. To improve resilience, over-allocation of nodes is an obvious avenue for future work. It is not entirely straightforward, due to topological constraints and heterogeneous node types.

Distributed nodes use an analogous disk loading mechanism. Each time a distributed node reboots, it does so from a CD-ROM which then negotiates with `masterhost` to, if necessary, securely apply software updates or reload the disk over the network. On each distributed node, Netbed instantiates a new Jail in a known initial state, analogous to the known initial state of a local node after disk loading and booting. In addition, a Jail can be "powered off" by terminating it or "rebooted" by restarting it.

Once a node or Jail has booted, our initialization sequence invokes a node configuration script that uses a program called the Testbed Master Control Client, TMCC, to securely communicate with a daemon on `masterhost` that fronts the database. Using this script and TMCC, a node obtains and initializes its hostname, experimental network IP addresses, routes, software packages, user accounts, and other configuration information. Local nodes NFS-mount the appropriate project tree and users' home directories from `fileserver`; in the wide-area, SFS is used instead.

3.6 Experiment Control

Traditional operating systems provide signals as a rudimentary form of control over local processes. Whereas users often start, stop, and resume processes, experimenters want to start, stop, and resume traffic generators and network monitors. To support dynamic experiment control, Netbed uses an event system to extend the notion of signals across sets of nodes and links. This facility closely mirrors the style of event schedulers found in network simulators. Just as simulation allows experimenters to manipulate link characteristics at prescribed times, so too can experimenters dynamically change latencies, bandwidths, and loss rates on emulated links. For example, to bring down a link named `link0` 10.5 seconds after experiment creation, a script would specify: `$ns at 10.5 "$link0 down"`.

Our event system is built on top of Elvin [38], a publish/subscribe system that supports federation. Static events are extracted from the database and fed into Elvin at experiment creation time. Dynamic events may be created through library interfaces and a command-line tool. Current clients of the event system include traffic generators, a WAN emulator control agent, a general remote execution facility, and Netbed's own management programs.

The event system is used extensively on local nodes but sparingly on distributed nodes, due to its current insecure deployment. Well-known solutions exist to secure the system; we are exploring a number of them, including using Elvin's "security keys," which limit the exchange of subscriptions and events to specific producers and consumers.

The event system controls high-level abstractions as defined in the *ns* configuration file, including links, nodes, and program objects. If experimenters were restricted to such high-level interfaces and tools, Netbed would limit the granularity of their control. Therefore, to the extent allowed by local policy, Netbed provides low-level and open access to resources, including root privileges on local nodes and Jail-restricted root privileges on distributed nodes. Of course, with such privileges experimenters can unwittingly corrupt their resources. Netbed's ability to quickly restore an experiment's hard state from the database and reload disk images makes it easy to recover from such accidents.

Root access on local nodes has proven to be an especially valued aspect of control, since it enables experiments requiring kernel modifications or access to raw sockets. To maintain security and isolation in the face of root access, Netbed prevents MAC and IP spoofing on local nodes through switch mechanisms. Since privileged access is mediated by Jail on shared, distributed nodes, these issues are not a concern there: though a process "in jail" can access raw sockets, it can only bind to its assigned IP address. This gives experimenters access to tools such as `tcpdump` and `traceroute`, without exposing insecurities.

Since the local nodes currently in use have serial console lines, power controllers, multiple network interfaces, and are dedicated to an experiment, they provide additional control mechanisms. Each local node is connected to a separate control network, isolated from the networks that are used for experimental traffic. This separate network provides three important features: more reliable control, cleaner experimental data, and greater security. Unless a program requires the use of a display or mouse attached directly to the node, Netbed does not penalize remote experimenters—with only minor exceptions, remote users have as much control over these nodes as they do over desk-side machines. For example, node consoles are virtualized so that an experimenter need not be logged into the server that physically hosts the serial console lines. Instead, all consoles can be securely accessed from any Unix or Windows machine via a local telnet session, connected through a transparent application-level SSL tunnel. We find that most kernel developers, once they have tried it, prefer remote use of Netbed machines to using desk-side test boxes.

3.7 Preemption and Scheduling

Traditional operating systems preempt and schedule processes for better system throughput and CPU utilization. Because Netbed manages shared community resources, efficient utilization is also a priority. Local nodes currently use a conservative allocation policy: each virtual node is mapped to a separate physical node. Therefore, Netbed can preempt idle experiments on local nodes to reacquire physical resources and to satisfy "runnable" experiments. Distributed nodes typically run each virtual node within a Jail, and are not currently subject to preemption. This policy is used because an idle distributed virtual node consumes only a single Jail rather than an entire physical node, and additional OS resource accounting mechanisms would be needed to accurately detect idle virtual nodes.

Local nodes are often idle despite being assigned to experiments. Determining idleness in Netbed is difficult; the indicators used in standard clusters are not sufficiently sensitive, since activity may constitute something as simple as infrequent network probes. Netbed's idle detection system currently monitors three metrics: traffic on the experimental networks, use of pseudo-terminal devices, and CPU load averages.

To avoid inconveniencing users, we manually confirm idle indications with them before swapping out their experiments. With recent tuning of the idle detection heuristics, Netbed has not experienced false positives and appears to find all truly idle experiments. Since our current swapping mechanism preserves only hard state, users with experiments dependent on soft state may manually disable preemption. With planned future work in disk state saving, Netbed should be able to safely preempt such experiments.

When experimenter interaction is not required, Netbed can fully automate the experimentation process by scheduling batch experiments, which execute whenever resources become available. Batch processing allows an experimenter to iterate over a large problem space without manual interaction. It also helps accommodate large experiments that may only find sufficient resources at low-usage, inconvenient times. Such off-peak scheduling further improves Netbed utilization.

4 Improving Network Experimentation

While Netbed provides most of the benefits of emulation, simulation, and wide-area experimentation, it is more than a simple sum of services. Netbed's common set of tools and abstractions have important practical benefits for experimentation, including: automated and efficient realization of virtual topologies, efficient use of resources through time- and space-sharing, and increased fault-tolerance through resource virtualization.

The savings afforded by automated mapping of a virtual topology to physical devices removes a significant experimentation barrier. Our user experiments show that after learning and rehearsing the task of manually configuring a 6-node "dumbbell" network, a student with significant Linux system administration experience took 3.25 hours to accomplish what Netbed accomplished in less than 3 minutes. This factor of 70 improvement and the subsequent programmatic control over links and nodes encourage "what if" experiments that were previously too time- and labor-intensive even to consider.

Efficient use of scarce and expensive infrastructure is also important and a sophisticated testbed system can markedly improve utilization. For example, analysis of 12 months of Netbed's historical logs gave quantitative estimates of the value of time-sharing (i.e., swapping out idle experiments) and space-sharing (i.e., isolating multiple active experiments). Although the behavior of both users and facility management would change without such features, the estimate is still revealing. Without Netbed's ability to time-share its 168 local Utah nodes, a testbed of 1064 nodes would have been required to provide equivalent service. Similarly, without space-sharing, 19.1 years, instead of one, would be required. These are order-of-magnitude improvements.

Netbed virtualizes node names and IP addresses such that equivalent nodes can be used interchangeably. For example, when an experiment is swapped in, it need not execute on the same set of physical nodes. Any nodes exhibiting the same properties and interconnection characteristics are suitable candidates. The flexibility to allocate from an equivalence class provides fault tolerance. If a node or link fails, an experimenter need not wait until the node or link partition is available again, but may instead re-map the experiment to an equivalent set of machines. This feature is valuable wherever node or link failures are anticipated, such as within large-scale clusters or wide-area networks.

5 Key Services and Evaluation

Much of *ns*'s popularity and power result from the flexibility it gives experimenters to efficiently change parameters and network scenarios. Netbed aims to bring a similar level of control and ease of use to emulated and

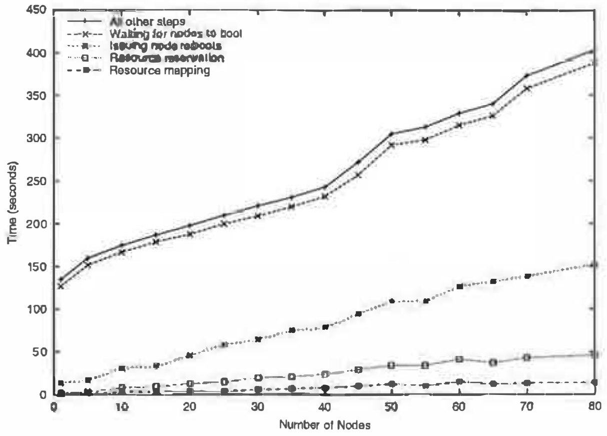


Figure 3: Time to create an experiment without disk loading. Times shown are cumulative, i.e., the difference between adjacent lines represents the time for that step.

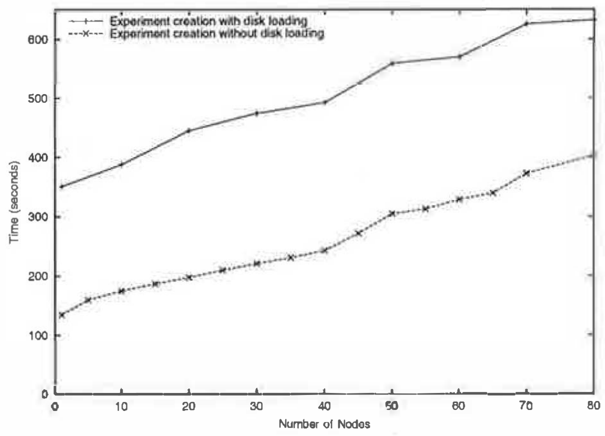


Figure 4: Time to create an experiment with disk loading. Time without disk loading from Figure 3 is also shown for comparison; note that the y-axis scale is different here.

wide-area experimentation, through automation and efficiency. In this section we describe the main challenges to Netbed's efficiency, and evaluate how well Netbed meets those performance challenges. These challenges include experiment creation and swapping, disk loading, mapping of virtual resources to local and distributed physical resources, and multiplexing simulated nodes.

5.1 Experiment Creation and Swapping

This subsection quantifies the time spent in experiment creation, which is comprised of parsing, global resource allocation, and local self-configuration, as described in Section 3. These results apply only to local resources; since distributed nodes are typically shared resources, Netbed does not routinely reboot them or re-install disk images on experiment creation. As shown in Figures 3 and 4, disk loading and node rebooting dominate experiment creation time. Therefore, configuration of distributed nodes is lightweight and not examined here.

The top line in Figure 3 shows the total time to create typical experiments. The duration of experiment creation is essentially equal to the swap-in duration, since the one-time expenses unique to experiment creation are insignificant compared to the cost of mechanisms shared by both, such as node rebooting. A single-node experiment takes 135 seconds. The majority of this time is spent rebooting the node and waiting for it to finish booting. As experiment sizes grow, creation time remains linear, with a marginal cost per node of approximately 3.4 seconds. Throughout the process, Netbed exploits parallelism as much as possible. For example, although it takes non-negligible time, VLAN setup does not contribute to creation time because it occurs in parallel with the longer node reboot stage.

Figure 3 also breaks out the costs of the most time-consuming stages of experiment creation, in the order those steps occur. The bottom line represents the time taken by `assign` to map physical resources. The next line is for reservation of those resources, which turns out to be dominated by reassigning serial console lines and logs. The next line is for issuing reboots to the nodes. They are rebooted in parallel, with a ten second pause every eight nodes so as not to over-stress network resources and lose too many control-related UDP packets, typically manifested by nodes failing to boot.[1] Finally, in the slowest step, Netbed waits for all nodes to come back up. The PC's BIOS is the biggest culprit; average time spent in the BIOS was 55 seconds for the nodes used in this experiment. Netbed also has 40 nodes that spend only 20 seconds in the BIOS, but in order to achieve consistency up to large scales, we limited these experiments to the more numerous nodes.

Figure 4 shows the additional expense of automatic disk loading, performed when an experimenter requests a custom disk image. Since our default dual-boot FreeBSD/Linux disk images prove sufficient for most experimenters, the majority of experiments do not incur this cost. Though much of the added time comes from transferring and writing the new disk image, a significant amount comes from rebooting each node twice (once to enter the disk loader, and again into the newly-loaded operating system). Although the absolute time for experiment creation is higher when loading disks, it is similarly scalable; the marginal cost per node is comparable.

5.2 Mapping Local Resources

Netbed's local assignment phase must not only realize user-specified node types, features, link characteristics, and topologies, but must also respect the limitations of available bandwidth. That is, Netbed ensures that the physical hardware will support the emulated traffic flows

[1]The PXE ROMs use UDP and a fixed timeout that we cannot change; hence we are forced to work around the problem.

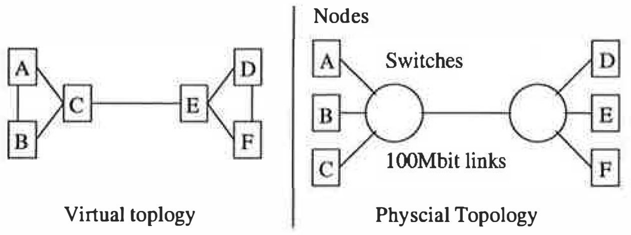


Figure 5: A trivial six-node partitioning problem

without introducing any bottlenecks, with their accompanying experimental artifacts. To map the desired virtual topology of Figure 5 onto the physical topology shown to its right, Netbed should pick a physical realization which groups A, B, and C together on one switch, and D, E, and F on the other switch; any other configuration will attempt to send excess traffic across the inter-switch link.

This *testbed mapping problem* problem is trivial in this six-node example, but in the general case, is NP-hard (by reduction to the multiway separator problem or the minimum-degree graph partitioning problem [13]). In conjunction with aggressive abstraction techniques to reduce the search space, `assign` uses simulated annealing [16], a randomized heuristic algorithm, to map virtual nodes and links to local nodes and VLANs. In addition to satisfying the individual experiment's requirements, the algorithm also attempts to minimize the required inter-switch bandwidth and the number of involved switches, in order to promote efficient utilization of the cluster.

Netbed has kept detailed logs of every experiment submitted since June 2001. We analyzed the following 12 months' data, covering over 2000 experiments. Figure 6 shows that a reliable indicator of the difficulty of a mapping problem, as measured by the runtime of `assign`, is the number of virtual nodes the user requests. We added a general notion of resource equivalence classes to `assign` in December 2001; the strikingly bimodal distribution in the figure demonstrates the resulting improvements. Grouping nodes into equivalence classes greatly reduces the search space since `assign` need only search the small number of equivalence classes rather than the large number of nodes. The new version takes less than 13 seconds on even the largest topologies and less than 5 seconds for most experiments.

5.3 Mapping Distributed Resources

The distributed case has different constraints. First, the underlying physical nodes are treated as fully connected, via the Internet. Second, distributed nodes are fairly well characterized by the nature of their "last-mile" link, e.g., cable modem, commodity Internet, or Internet2. Therefore, Netbed assigns corresponding intuitive subtypes to distributed nodes, e.g., `pc-cable`, `pc-inet`, `pc-inet2`. This typing lets experimenters request virtual

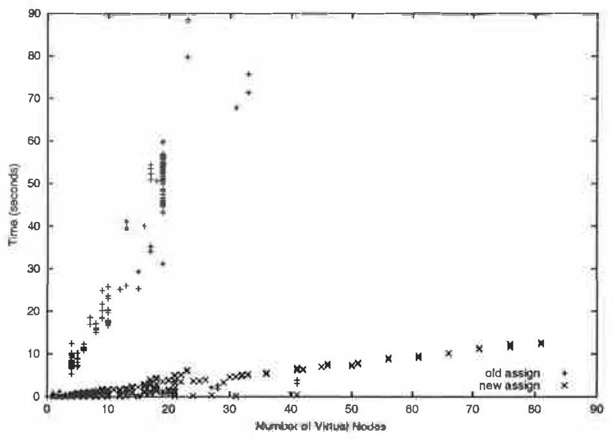


Figure 6: Performance and Scaling of `assign`

nodes by their type or subtype, rather than specify a particular topology connecting them. Netbed's generic resource assignment code, identical for both local and distributed resources, handles this common situation.

However, some experimenters may want more precisely-matched resources or a particular virtual topology. Netbed allows them to request a virtual topology with wide-area links of specific latency, loss, and bandwidth characteristics. They may assign weights to each of the three attributes, based on their perceived importance. Unlike the highly configurable local links in a Netbed cluster, connections between distributed nodes traverse the Internet through uncontrollable links. Therefore, our challenge is to map virtual nodes to physical resources such that the requested links best match the actual characteristics of the corresponding inter-node Internet paths. (Netbed's database is updated frequently with the measured latency and loss on the NxN paths, and occasionally updated with bandwidth measurements.)

This mapping is a variation of the NP-hard Quadratic Assignment Problem. To provide an efficient, best-effort solution, Netbed's `wanassign` is implemented as a genetic algorithm [39]. Possible solutions are scored based on how closely they match desired link characteristics. For each solution, a normalized sum of errors-squared is found for latency, loss rate, and bandwidth. A geometric mean of the three errors results in an overall score. `Wanassign` evolves its answer by propagating solutions with the least error.

We conducted two experiments to test `wanassign`'s performance. The first mapped a wide variety of virtual topologies onto a set of 16 physical, distributed nodes. We varied the number of requested nodes from 4 to 16 and the number of requested links from 4 to 120, examining 48 pairs from this set to present a cross section of experiment complexities. For each of these pairs, we ran hundreds of tests on automatically-generated topolo-

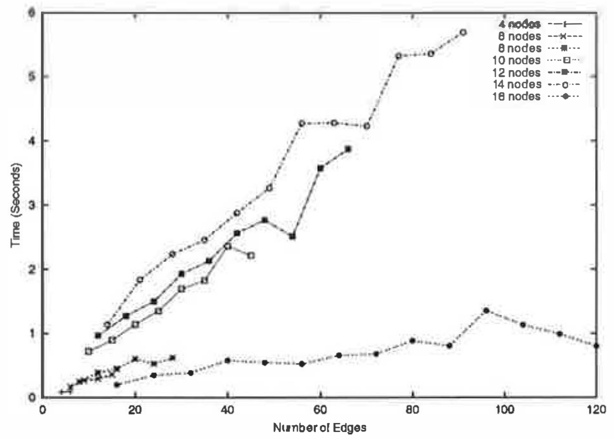


Figure 7: Average time for `wanassign` to find a solution for a variety of experimental topology complexities (node and edge counts).

gies. Figure 7 shows the average time to find a solution for each complexity. Interestingly, mappings using all 16 nodes were found much faster than mappings using most, but not all, of the nodes. The results show that for modestly-sized experiments, the algorithm does not contribute noticeably to the total experiment setup time, nor is it prohibitively slow for experiments involving most of the available nodes.

The second experiment explored further scalability, mapping a range of virtual topologies onto a synthetic set of 256 distributed nodes. All experiments requesting 32 virtual nodes, as well as all sparse topologies, mapped in a few minutes. For larger and denser topologies, up to 256 nodes and approximately 40 edges/node, mapping time ranged from 10 minutes to 2 hours. We expect to improve these results by an order of magnitude using the following three techniques: less stringent and more clever termination conditions; standard optimization techniques, in particular memoizing; and parallelizing the algorithm, which is practical in either a shared memory multiprocessor or on a cluster [39]. Finally, we expect major additional improvement to come from "binning" the nodes and links into groups with similar characteristics, dramatically reducing the search space.

5.4 Disk Reloading

An important feature of testbed control is the ability to reload the contents of node local disks automatically. This not only ensures node integrity, but also allows custom OS configurations. The two common approaches for achieving this goal are to load complete disk images [14, 32, 35] or to work through the file system to incrementally synchronize a target hierarchy with a reference copy (`rsync` [37], `Unison` [41]). There are five reasons for preferring disk imaging. (1) While sometimes more efficient in terms of network bandwidth, on our images, at least, the synchronization approach is slower. `rsync` takes over 50% longer to compare file timestamps on our typical image (80K inodes, 500MB data) than Netbed's disk loader takes to copy all the allocated blocks. Comparing hashes of file contents takes much longer. (2) Approaches that rely solely on file timestamps cannot be used for security reasons, as falsified timestamps allow modified files to corrupt the next experiment. (3) Approaches working through the file system cannot be used on corrupt target file systems, nor (4) to install custom OS's with unknown file systems. (5) Bulk disk imaging is scalable through multicast-based approaches. A third approach based on content hashes of blocks, as in LBFS [23], may be worth investigating.

Policy: The policy for disk reloading presents a tension between the latency of typical experiment creation, overall Netbed throughput, Netbed system complexity, node robustness, and experiments' security. Our policies have evolved over time, driven by our tools, pressure on resources, and experience.

Each node in a new experiment requires a clean disk. However, disk reloading remains the most time-consuming aspect of experiment creation and swap-in, even though we have reduced it to less than 100 seconds. Netbed's current policy reloads each node's disk with the default image containing both FreeBSD and Linux. This works well since most users request one of these OSes, and if there are sufficient free nodes, the disks are reloaded in the background and are immediately available for the next swap-in.

A troubling effect occurs, however, in the common case of a single experimenter creating and tearing down very similar experiments, in quick succession; this frequently also happens with the batch queue. The nodes are not available for the few (typically wasted) minutes while reloading, during which time the user requests a similar number of nodes for their next experiment. To avoid this anomaly we currently pace the reloading of freed nodes, instead of reloading them all at once. For security reasons, we allow an un-reloaded node to be assigned only to an experiment in the same project as the node's previous experiment. This approach, however, has robustness vulnerabilities, since the disk's soft state will not be reinitialized, and may have been changed by the previous experiment—though that is rare.

Users can also specify an alternate disk image or partition. In this case, the background disk reloading is wasted, as the default image is overwritten by the user's custom one. Automated analysis of historical and ongoing experiment creation and swap patterns is one promising way to attack this challenge.

Process: The procedure for disk reloading follows the initial steps described in Section 3.5: the PXE BIOS loads the initial bootstrap which in turn loads a small, memory file system-based FreeBSD system used to run the disk loader client. This client contacts an instance of the disk loader server, downloading, uncompressing and writing out the disk image. After completion, the node reboots from the newly installed image.

We currently provide a small set of images containing various versions of Linux and FreeBSD; we will soon add Windows XP. Custom disk images can be used to boot an unsupported OS, to load a newer (or older) version of a supported OS, or to install a specialized version of an existing image on multiple nodes.

The Netbed disk loader, termed "Frisbee" (the flying disk) uses three main techniques to improve performance from Netbed's first loader, which took 29 minutes per image. First, it carefully overlaps block decompression and device I/O. Second, it uses a domain-specific compression algorithm that uses file system information to identify which parts of the disk need to be saved; it compresses these portions with standard `zlib`-based compression. Third, it uses a custom reliable multicast protocol to deliver compressed images to clients, dramatically reducing the required server bandwidth and improving scalability. The result is that a standard FreeBSD image requires 88 seconds to load onto a single node. It also scales well; 80 nodes can be loaded simultaneously with an average of only 97 seconds per node, and with all nodes completing in 117 seconds. Frisbee's performance also compares favorably to commercial tools; in our initial tests, it was able to load our standard Linux image on a single node in 77% of the time taken by Norton Ghost.

The compression algorithm exploits the fact that many disks contain large swap partitions and mostly-empty file systems, and looks at partition types and file system free-block lists to find these. For example, one of our standard FreeBSD images for a 3GB partition is over 80% unused, and reduces to 156MB using Frisbee image compression, versus 473MB using naive `zlib` compression. In addition to saving network bandwidth when transferring the file, the file system-specific compression enables the Frisbee decompression program to optionally skip, rather than zero, the free file system blocks when writing the disk image. This turned out to be very important: once we had done standard compression and implemented a multicast mechanism, writing to the disk became the bottleneck. For the aforementioned FreeBSD disk image, Frisbee wrote 550MB of actual decompressed data rather than the full 3GB.

5.5 Scaling of Simulated Resources

Experiments can leverage simulation to multiplex simulated nodes onto a single physical node and to obtain greater scalability. Since the simulator interacts with the physical world through *nse*, it must keep pace with real time. Its ability to do so is dependent on the rate of events that need to be processed, rather than the number of nodes or links per se. Towards achieving greater scale, we have made several improvements and contributed fixes to *nse*. We describe here a simple study that achieves greater scale through simulation.

An instance of *nse* simulated 2Mb constant bit rate UDP flows between pairs of nodes on 2Mb links with 50ms latencies. To measure *nse*'s ability to keep pace with real time, and thus with live traffic, a similar link was instantiated inside the same *nse* simulation, to forward live TCP traffic between two physical Netbed nodes, again at a rate of 2Mb. On an 850MHz PC, we were able to scale the number of simulated flows up to 150 simulated links and 300 simulated nodes, while maintaining the full throughput of the live TCP connection. With additional simulated links, the throughput dropped precipitously. We also measured *nse*'s TCP model on the simulated links: the performance dropped after 80 simulated links due to a higher event rate from the acknowledgment traffic in the return path.

More complex hybrid topologies exposed unanticipated routing behavior. Incorrect routing arises when an *nse* simulation, running on a multihomed host, relies on its kernel's routing tables. The solution required Netbed's global system perspective; it computes the overall routes, using Unix policy routing mechanisms (`ipfw` and `ipchains`) to control the packet routes.

6 Validation and Testing

This section validates Netbed's emulation capabilities through micro- and macro-benchmarks. Since Netbed is itself a complex and evolving distributed system, it requires continual testing and validation. This section therefore outlines a testing methodology intended to ensure Netbed's continued accuracy.

6.1 WAN Emulator Validation

There are two concerns with using off-the-shelf PCs and a general purpose operating system for emulation: first, machines must be able to keep pace when emulated links are operating at full speed; second, delays, bandwidths, and packet loss rates should be emulated accurately.

Emulation nodes in Netbed run a FreeBSD 4.6 kernel with Dummynet and polling device drivers. We run these kernels with a clock frequency of 10000HZ to allow sub-millisecond delay granularity, while the polling drivers reduce interrupt load and provide improved precision.

As a capacity test, we generated streams of UDP round-trip traffic between two nodes, with and without an interposed emulator node. The emulator node showed no adverse effects on 1518-byte packets; either configuration easily saturated a 100Mb link. With 64-byte packets, the two nodes exchanged 55000 packets (3.5MB) per second when connected directly versus 37000 packets

delay (ms)	packet size	observed Dummynet RTT	stdev	% err	adjusted Dummynet RTT	% err	observed *nse* RTT	stdev	% err	adjusted *nse* RTT	% err
0	64	0.177	0.003	N/A	N/A	N/A	0.238	0.004	N/A	N/A	N/A
	1518	1.225	0.004	N/A	N/A	N/A	1.554	0.025	N/A	N/A	N/A
5	64	10.183	0.041	1.83	10.006	0.06	10.251	0.295	2.51	10.013	0.13
	1518	11.187	0.008	11.87	9.962	0.38	11.586	0.067	15.86	10.032	0.32
10	64	20.190	0.063	0.95	20.013	0.06	20.255	0.014	1.28	20.017	0.09
	1518	21.185	0.008	5.92	19.960	0.20	21.675	0.093	8.38	20.121	0.61
50	64	100.185	0.086	0.18	100.008	0.00	100.474	0.029	0.47	100.236	0.24
	1518	101.169	0.013	1.16	99.943	0.05	102.394	3.440	2.39	100.840	0.84
300	64	600.126	0.133	0.02	599.949	0.0	601.690	0.546	0.28	601.452	0.24
	1518	600.953	0.014	0.15	599.728	0.04	602.999	0.093	0.49	601.445	0.24

Table 1: Accuracy of Dummynet and *nse* delay at maximum packet rate as a function of packet size and link delay. The 0ms measurement represents the base overhead of the link. Adjusted RTT is the observed value minus the base overhead.

bandwidth (Kbps)	packet size	observed Dummynet bw (Kbps)	% err	observed *nse* bw (Kbps)	% err
56	64	56.06	0.11	55.60	0.71
	1518	56.67	1.89	56.63	1.12
384	64	384.2	0.05	376.3	2.00
	1518	385.2	0.34	382.1	0.49
1544	64	1544.7	0.04	1444.5	6.44
	1518	1545.8	0.11	1531.0	0.84
10000	64	10004	0.04	N/A	N/A
	1518	10005	0.05	9659.6	3.40
45000	1518	45019	0.04	39857	11.43

Table 2: Accuracy of Dummynet and *nse* bandwidth as a function of link bandwidth and packet size.

packet loss rate (%)	packet size	observed Dummynet loss rate (%)	% err	observed *nse* loss rate (%)	% err
0.8	64	0.802	0.2	0.819	2.37
	1518	0.803	0.3	0.820	2.50
2.5	64	2.51	0.4	2.477	0.92
	1518	2.47	1.1	2.477	0.92
12	64	12.05	0.4	11.88	1.00
	1518	12.09	0.7	11.89	0.91

Table 3: Accuracy of Dummynet and *nse* packet loss rate as a function of link loss rate and packet size.

(2.4MB) when joined by an emulator node. Since these are round trip measurements, the packet rates are actually twice the numbers reported.

To bound the accuracy and precision of emulation nodes, we performed a series of experiments using a representative range of delay, bandwidth, and packet loss rate values coupled with high packet rates for both large and small packets.

After establishing maximum emulation rates for large and small packets, we ran a series of tests using those packet rates with various delay, bandwidth, and loss rate values, measuring both accuracy and precision. The delay results are presented in Table 1. The 0ms rows represent the base overhead associated with interposition of an emulation node. These results seem to indicate, and further experimentation confirmed, that emulation node overhead is proportional to the packet size. As indicated in the "observed" column, small packets show noticeable error with delays less than 10ms and large packets suffer with delays less than 50ms. While both are tolerable for wide-area emulation, we can improve accuracy by adjusting delays to compensate for emulation overhead. As a first approximation, we scaled delays by the base overhead shown in the 0ms case. The adjusted results, shown in the "adjusted" column, are both accurate and precise.

To measure the bandwidth limiting capabilities of an emulation node, we used one-way traffic. A sender node sent packets through an emulation node to a consumer node, which calculated bandwidth. Results are summarized in Table 2.

Finally, using the same setup, we instead measured packet loss rates as observed by the consumer. Results are summarized in Table 3.

6.2 *nse* Validation

This section uses the methodology of Section 6.1 to validate the observed latencies, bandwidths, and loss rates induced by *ns*'s emulation facility, *nse*, against their expected values. *nse* runs on a FreeBSD 4.5 kernel at 1000HZ. The simulation is configured with two nodes and a duplex link connecting them. The physical node running *nse* interposes two other traffic-generating physical nodes. This setup mimics Section 6.1, differing only in packet rate. A maximum stable packet rate of 4000 packets per second was determined over a range of packet rates and link delays using 64-byte and 1518-byte packets. Note that the actual capacity is twice this value due to the duplex link. With this capacity, we performed experiments to measure the delay, bandwidth and loss rates for representative values. The results are summarized in Tables 1, 2 and 3. Netbed's integration of *nse* is much less mature than its support for Dummynet. This is reflected in the larger relative error rates of *nse* bandwidth and loss rates with respect to Dummynet. Integrating *nse* has already uncovered a number of problems that have since been solved; as we continue to gain experience with *nse*, we expect the situation to improve.

	Live Internet			Emulated		
	tics	stddev	retransmits	tics	stddev	retransmits
Fast	29	0.00	1.10	28	0.67	1.10
Slow	21	0.73	1.70	21	0.52	2.80

Table 4: Median "tic" rates and packet retransmission counts achieved by DOOM clients, both on live Internet and emulated links. Numbers are repeated both for nodes with uniformly fast links and with some intermixed slower links.

6.3 Validation Against a Wide-Area Network

This section validates Netbed's emulation mechanisms against a wide-area network: it compares two macro-benchmarks run on a set of live Internet nodes and then within a corresponding emulation. The first example also demonstrates the transparency of Netbed's heterogeneous resource specification and its ability to provide a best-fit mapping between requested wide-area links and live Internet links.

Distributed Multiplayer Game: This benchmark evaluates a derivative of DOOM on four network configurations, making at least four repeated runs on each. In these scenarios, five synthetic clients communicate using a simple protocol. At a target rate of 30 times per second, each client sends unicast packets to all other clients, doing so only after receiving all packets from the prior period. We specified the desired latency and bandwidth of the ten links comprising a fully-connected graph between the five clients.

The first configuration specified a node type of `pcvremote` to obtain wide-area "virtual" nodes. In this sense, virtual means the nodes may be multiplexed onto a single physical distributed node. Netbed's distributed mapping service, the genetic algorithm described in Section 5.3, found the best-matching fit from among the distributed nodes with available virtual node "slots."

The second configuration used the same link specification, but instead of mapping to the live Internet, requested emulation on local nodes and links. Making that switch to an entirely different experimental environment required changing only one line within a Tcl loop that set the node type. The third and fourth configurations were analogous to the first two configurations, but requested a few substantially slower links.

The results were similar between emulation and the live Internet, as presented in Table 4. The two key metrics in DOOM are "tic rate" and packet retransmission. Tic rate in this example is affected primarily by latency, and represents the rate at which progress is made in the system—a higher tic rate indicates faster progress. Packet retransmission rates are governed by bandwidth and packet loss rate; there are typically only a handful of retransmitted packets per trial.

Wide-Area Database Replication: Researchers at Johns Hopkins University are studying group communication mechanisms for wide-area replication of databases. In the course of their research, they compared results from the CAIRN wide-area network [7] to those obtained emulating the observed CAIRN delay and bandwidth characteristics with Netbed. Their application-level measurements of communication characteristics matched well [3]. Netbed offered two advantages over CAIRN: First, with Netbed's control, they were able to study the system-wide effects caused by varying network characteristics. Second, they were able to obtain a set of nodes of a consistent type.

6.4 Testing

Netbed presents unusual testing challenges: First, it is inherently coupled to physical artifacts which, unlike software state, can not be cloned. This makes full test and regression runs impossible. Second, its mission is to provide a public evaluation platform for arbitrary programs. This mission simultaneously puts a premium on accuracy and precision, while presenting a fundamentally unknowable workload. Combined, these two reasons also mean that Netbed must run continuously, even as its software radically evolves.

We have countered with the following procedures. First, we have created a separate 8-node Netbed, Minibed. As an independent Netbed instance, Minibed is also important to our future work on federation.

Second, we have integrated support for testing throughout the Netbed software suite. In addition to the normal operating mode, all of our software supports a "test mode" in which any operations that normally affect hardware are prevented. It allows us to make duplicate installations of Netbed databases and software, including web interfaces and daemons, and to run tests of the software without requiring exclusive access to hardware. We also have incorporated a "full-test mode," in which we can reserve hardware in the master Netbed database and use that hardware in conjunction with the duplicate database and software. This enables the test environment to affect this hardware, which is ignored by the "main" Netbed system. This feature is made possible by database-driven, node-specific redirection to alternate daemons and databases.

Third, we have developed a comprehensive regression test suite that is run nightly and optionally at compile time. However, we currently only systematically test for software bugs. To monitor Netbed accuracy, we are adding additional point tests as well as end-to-end tests.

7 New Experimental Techniques

This section showcases the novel experimental opportunities made possible by Netbed. The first case study capitalizes on Netbed's *ns* compatibility to automate comparison of emulated and simulated results. Other systems have leveraged a similar synergy between simulation and live experimentation [6], but required adoption of a non-standard programming interface. The second case study shows the importance of automation.

7.1 TCP Dynamics

Network simulators, such as *ns*, have proven invaluable in studying TCP behavioral dynamics [11]. Nevertheless, with its abstractions such as one-way protocols with simplified window and ACK behavior, simulation should be validated empirically. Ironically, the potential for bugs and unspecified design parameters mean that real implementations do not necessarily define valid behavior, either. Fortunately, the notion of "deviant behavior" [9] allows an experimenter simultaneously to gain confidence in the validity of simulation and the correctness of implementation. This case study leverages existing simulation experiments to drive emulated scenarios. This approach makes an existing corpus of test scenarios amenable to live experimentation. Thus, corner cases with known results can be applied as regression tests to real network stacks to evaluate their conformance.

The *ns* maintainers run nightly regression tests [24]. Netbed's ability to parse *ns* scripts means these scripts can instead be used to validate *ns* behavior against emulation. Further, the tests may drive regression testing of a kernel implementation or a comparison across several implementations. This section presents preliminary results that show the feasibility of automating this process. The study of low-level, fine-grained TCP dynamics shows Netbed's flexibility in modulating a virtual network at various scales.

Our framework executes a test script within *ns* and parses output trace files to determine where to generate traffic, which packets are dropped, and which links suffer losses. It then configures a network topology via Netbed's event system and passes a list of target drop packets to the correct Dummynet node (we have extended Dummynet to drop packets by ordinal packet number). Again via the event system, the framework starts a program object to record packet traces and finally invokes the traffic generators.

Figure 8 shows a simple test from the *ns* validation suite that drops a single packet in a TCP New Reno stream. The *ns* and FreeBSD 4.5 senders detect a Triple Duplicate ACK and perform a Fast Retransmit immediately. They behave similarly; over 10 experiments FreeBSD 4.5 achieves a mean throughput of 50232Bps (standard deviation 4.09) and *ns* achieved 48090Bps. By contrast, we discovered that FreeBSD 4.3 does not retransmit until triggered by a timer expiration, which greatly degrades throughput. The behavior in FreeBSD 4.3 is caused by an uninitialized variable. A thorough application of the full suite of TCP tests may well uncover additional subtle bugs that would be exceedingly difficult to detect and reproduce without Netbed's fine-grained control.

7.2 The Armada I/O Framework

Simulation allows an experimenter to effortlessly explore a large parameter space. Using Netbed's programmatic *ns* interface to loop over a configuration space and exercising its distributed event system to affect link characteristics, an experimenter has similar power over emulation. Oldfield and Kotz [30] used these techniques in evaluating Armada [29], a file system for computational grids. Armada's performance is highly dependent on link bandwidth, latency, and packet loss rate. The authors used Netbed's batch system to evaluate every possible combination of 7 bandwidths, 5 latencies, and 3 application parameter settings on four different configurations on a set of 20 nodes, performing a total of 420 different tests in 30 hours, averaging 4.3 minutes each.

8 Related Efforts

Network Emulation: ModelNet [42] is a new network emulation system focused on scalability. It uses a small gigabit cluster, running a much extended version of Dummynet, which is able accurately to emulate an impressively large number of moderate speed links. This core routes packets between applications running on additional "edge nodes." Applications can be multiplexed on edge nodes, without resource isolation. ModelNet shares some of Netbed's automatic configuration of physical resources by including tools to take a target topology specified in a high-level format and map it into ModelNet mechanisms; it provides the added capability of optionally distilling the topology to trade accuracy for scalability.

ModelNet emphasizes scalability through a high-performance implementation of emulated links. This contrasts with our emphasis on complete accuracy through conservative resource allocation, exposure of all resources (including link emulation mechanisms) to manipulation by experimenters, and integration of disparate techniques into a common framework.

ModelNet's core contributions are complementary to Netbed's; indeed, we intend to work together to integrate ModelNet into Netbed. This combination should bring Netbed's rich user interface and ease of use to ModelNet, while adding a scalable new mechanism to those available through Netbed's common abstractions.

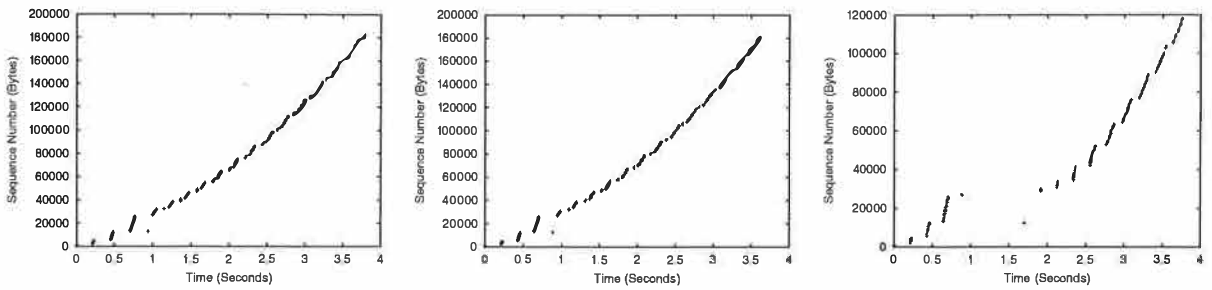


Figure 8: New Reno One Drop Test: (a) *ns* (b) FreeBSD 4.5 (c) FreeBSD 4.3 (different y-axis scale)

Yet another link emulation technique is trace modulation [27], which recreates observed end-to-end characteristics of a wireless network. Interposing trace modulation instead of Dummynet would bring wireless emulation to Netbed.

There have been a large number of single-node network emulation efforts. These include hitbox [1], ONE [2], NIST Net [26], and Rice's support for evaluating their OS optimizations [31]. Another category is represented by the "Orchestra" fault-injection system [8]. With a few exceptions, these single node emulators were tailored for a specific research application. A few multi-node network emulators have been planned or built, but only for specific projects. One of the earliest and largest was a particular configuration of 12 workstations at USC in 1994, used to study TCP Vegas [1]. They cite an emulator effort at Bell Labs [19], which apparently started to build a more general emulator.

Distributed Network Testbeds: The "Access" vision [5] originated the idea of a set of small testbeds, distributed over dozens of sites. The Access vision overlapped with Netbed in our shared emphasis on completely replaceable node software and our operational model of a Web-accessible master control host. However, Access did not intend to provide an emulation facility nor did it intend to offer integration. They did recognize a need we identified only later, for real wide-area links for some experimenters.

PlanetLab [33] is a new effort that plans to provide to researchers a large number (1000) of centrally administered, geographically distributed PCs, along with a modest number of clusters. This testbed, currently in its initial phase, would be used for arbitrary research, yet provide a transition avenue to production deployment of overlay network services. Unlike Netbed, PlanetLab plans to emphasize the design of APIs and services that can be shared by higher-level services.

Netbed's distributed node support is similar to what is planned for PlanetLab's next phase. Although with a different primary goal, PlanetLab's notion of a "service" across a "slice" of PlanetLab nodes is similar to Netbed's "experiment," since Netbed experiments can be of arbitrary duration. An experiment is richer in that it contains flexible notions of topology, swapping, hard state, soft state, and optional shared persistent storage. Like Netbed, PlanetLab's current testbed management is centralized. Their future plans emphasize unbundled management in order to facilitate research into management; our plans emphasize federation, in order to achieve greater scalability and another route to overlay service deployment. In fact, we are jointly exploring providing access to PlanetLab through Netbed's interface.

Network Simulators: Network simulators successfully isolate protocol dynamics but may do so at the expense of accuracy. Therefore, results from simulators may not be valid indicators of deployed performance [11]. Brakmo and Peterson [6] highlight differences between simulated and implemented TCP protocols. Their *x*-kernel-based simulator avoids inaccuracies by using actual protocol code, as does recent work integrating Click elements into *ns* [25]. However, both systems rely on non-standard protocol implementations.

Cluster Management: Through its virtualization of cluster hardware and software, "Emulab Classic"—Netbed's cluster-based emulation portion that has been in public production use since October 2000—is relevant far beyond network experimentation. In its flexible and efficient allocation of all hardware and software resources (except shared persistent storage) and ability to isolate virtual sub-clusters, Emulab overlaps many or most of the low level facilities in "computing utility" efforts such as IBM's Océano [28], HP's Utility Data Centers, and Duke's Cluster-on-Demand [22]. Netbed has the flexible interfaces and all the needed mechanisms—including dynamically adding or removing nodes in an experiment—to support reconfiguration by Service Level Agreements or by sub-cluster management systems.

9 Conclusion

Acting as a virtual machine for network experimentation, Netbed virtualizes and integrates simulated, emulated, and distributed nodes and links. Through a rich user interface, efficiency, and automation, Netbed enables qualitatively new kinds of experimentation across these mechanisms.

References

[1] J. S. Ahn et al. Evaluation of TCP Vegas: Emulation and Experiment. In *Proc. of SIGCOMM '95*, pages 185–195, Aug. 1995.

[2] M. Allman, A. Caldwell, and S. Ostermann. ONE: The Ohio Network Emulator. Technical Report TR–19972, Ohio University Computer Science, Aug. 1997.

[3] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. Practical Wide-Area Database Replication. Technical report, Johns Hopkins University, 2002.

[4] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th SOSP*, Oct. 2001.

[5] T. Anderson. A Case for Access: A High Performance Communication and Computation Environment for Wide Area Distributed Systems, Networking, and Applications Research. http://-www.cs.washington.edu/homes/tom/access/.

[6] L. S. Brakmo and L. L. Peterson. Experiences with Network Simulation. In *Proc. of ACM SIGMETRICS'96*, May 1996.

[7] CAIRN: Collaborative Advanced Internet Research Network. http://www.isi.edu/CAIRN/.

[8] S. Dawson et al. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In *Proc. FTCS '96*.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proc. 18th SOSP*, Oct. 2001.

[10] K. Fall. Network Emulation in the Vint/NS Simulator. In *Proc. IEEE ISCC '99*, 1999.

[11] S. Floyd and V. Paxson. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4), August 2001.

[12] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *Proc. 16th SOSP*, pages 38–51, Oct. 1997.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness.* W. H. Freeman, 1979.

[14] Symantec Ghost. http://www.symantec.com/sabu/ghost/.

[15] J. Heidemann et al. Effects of Detail in Wireless Network Simulation. http://www.isi.edu/~johnh/PAPERS/Heidemann00d.html.

[16] L. Ingber. Very Fast Simulated Re-Annealing. *Journal of Mathematical Computer Modelling*, 12:967–973, 1989. http://www.-ingber.com/asa89_vfsr.ps.gz.

[17] IXP1200. http://www.intel.com/design/network/products/-npfamily/ixp1200.htm.

[18] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proc. 2nd Intl. SANE Conference*, May 2000.

Acknowledgments: Many thanks to Chris Alfeld, Dave Andersen, and Kirk Webb for discussion, design, and code, to Dave for MIT's RON testbed nodes, to Russ Christensen, Alastair Reid, Tim Stack, and Parveen Patel for running experiments, to Eric Eide for editing, to ex-Fluxers who helped build the Emulab cluster, to John Regehr, Robert Morris, and the anonymous reviewers for comments, to Jim Griffioen for bravely being the first to bring up another cluster, to Ron Oldfield for the Armada results, to Nicolas Christin for suggesting validating TCP dynamics, to Mark Tinguely for clarifications on FreeBSD TCP, and to our many users. Finally, we are grateful to our many sponsors, especially NSF under grants ANI-0082493 and ANI-0205702, Cisco Systems, and DARPA/Air Force under grant F30602-99-1-0503.

[19] A. M. Lapone, N. F. Maxemchuk, and H. Schulzrinne. The Bell Laboratories Network Emulator. Technical Report BL01113820-930913-64TM, AT&T Bell Labs, Sept. 1993.

[20] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. of SOSP '99*, December 1999.

[21] P. E. McKenney, D. Y. Lee, and B. A. Denny. *Traffic Generator Software Release Notes.* SRI International and USC/ISI Postel Center for Experimental Networking. http://www.postel.org/tg/.

[22] J. Moore and J. Chase. Cluster On Demand. Technical Report CS-2002-07, Duke University, Dept. of Computer Science, May 2002.

[23] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-bandwidth Network File System. In *Proc. 18th SOSP*, Oct. 2001.

[24] The Network Simulator ns-2: Validation Tests. http://www.isi.-edu/nsnam/ns/ns-tests.html.

[25] M. Neufeld, A. Jain, and D. Grunwald. Nsclick: Bridging Network Simulation and Deployment. In *Proc. MSWiM 2002.*

[26] NIST Internetworking Technology Group. NIST Net home page. http://www.antd.nist.gov/itg/nistnet/.

[27] B. D. Noble et al. Trace-Based Mobile Network Emulation. In *Proc. of SIGCOMM '97*, Sept. 1997.

[28] Océano Project. http://www.research.ibm.com/oceanoproject/.

[29] R. Oldfield and D. Kotz. Armada: A parallel file system for computational grids. In *Proc. of IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2001.

[30] R. Oldfield and D. Kotz. Using the Emulab network testbed to evaluate the Armada I/O framework for computational grids. Technical report, Dartmouth, May 2002. ftp://ftp.cs.dartmouth.-edu/pub/raoldfi/armada/oldfield:armada-emulab-tr.pdf.

[31] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *Proc. 3rd OSDI*, Feb. 1999.

[32] Partition Image. http://www.partimage.org/.

[33] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. HotNets-I*, Princeton, NJ, Oct. 2002.

[34] PXE Preboot Execution Environment Specification Version 2.1. ftp://download.intel.com/ial/wfm/pxespec.pdf.

[35] Rembo Technology. BpBatch. http://www.bpbatch.org/.

[36] L. Rizzo. Dummynet and Forward Error Correction. In *Proc. of the 1998 USENIX Annual Technical Conf.*, June 1998.

[37] rsync. http://rsync.samba.org/.

[38] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. In *Proc. AUUG '00*, June 2000.

[39] R. Tanese. The Distributed Genetic Algorithm. In *Proc. ICGA '89.* Morgan Kaufmann, 1989.

[40] The VINT Project. *The* ns *Manual*, Apr. 2002. http://www.isi.-edu/nsnam/ns/ns-documentation.html.

[41] Unison. http://www.cis.upenn.edu/~bcpierce/unison/.

[42] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. 5th OSDI*, Dec. 2002.

Scalability and Accuracy in a Large-Scale Network Emulator*

Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan,
Dejan Kostić, Jeff Chase, and David Becker
Department of Computer Science
Duke University
{*vahdat,grant,walsh,priya,dkostic,chase,becker*}*@cs.duke.edu*

Abstract

This paper presents ModelNet, a scalable Internet emulation environment that enables researchers to deploy unmodified software prototypes in a configurable Internet-like environment and subject them to faults and varying network conditions. Edge nodes running user-specified OS and application software are configured to route their packets through a set of ModelNet core nodes, which cooperate to subject the traffic to the bandwidth, congestion constraints, latency, and loss profile of a target network topology.

This paper describes and evaluates the ModelNet architecture and its implementation, including novel techniques to balance emulation accuracy against scalability. The current ModelNet prototype is able to accurately subject thousands of instances of a distrbuted application to Internet-like conditions with gigabits of bisection bandwidth. Experiments with several large-scale distributed services demonstrate the generality and effectiveness of the infrastructure.

1 Introduction

Today, many research and development efforts investigate novel techniques for building scalable and reliable Internet services, including peer-to-peer networks [17, 19, 20], overlay networks [1, 8], and wide-area replication [10, 23]. These projects and many others run on large numbers of cooperating nodes spread across the Internet. To test and evaluate such systems, their developers must deploy them in realistic scenarios, i.e., large, structured, dynamic wide-area networks.

* This work is supported in part by the National Science Foundation (EIA-9972879), Hewlett Packard, IBM, Intel, and Microsoft. Vahdat is also supported by an NSF CAREER award (CCR-9984328).

Unfortunately, it is difficult to deploy and administer research software at more than a handful of Internet sites. Further, results obtained from such deployments "in the wild" are not reproducible or predictive of future behavior because wide-area network conditions change rapidly and are not subject to the researcher's control. Simulation tools such as *ns* [15] offer more control over the target platform, but they may miss important system interactions, and they do not support direct execution of software prototypes.

This paper advocates *network emulation* as a technique for evaluating Internet-scale distributed systems. Network emulators subject traffic to the end-to-end bandwidth constraints, latency, and loss rate of a user-specified target topology. However, emulation has typically been limited to systems that are relatively small and static.

We present the design, implementation, and evaluation of ModelNet, a scalable and comprehensive large-scale network emulation environment. In ModelNet, unmodified applications run on *edge nodes* configured to route all their packets through a *scalable core* cluster, physically interconnected by gigabit links. This core is responsible for emulating the characteristics of a specified target topology on a link-by-link basis, capturing the effects of bursty flows, congestion, etc. In this context, this paper makes the following contributions:

- We show that a single modern server-class PC can accurately emulate bandwidth, latency, and loss rate characteristics of thousands of flows whose aggregate bandwidth consumption in the target topology approaches 1 Gb/s. Depending on communication patterns and topology mapping, we are able to scale available bisection bandwidth linearly with additional core nodes in our switched gigabit network hosting environment.

- We demonstrate the utility of a number of techniques that allow users to trade increased emulation scalability for reduced accuracy. This is required because, in general, it is impossible to capture the full complexity of the Internet in any controlled environment. Sample approaches include: i) progressively reducing the complexity of the emulated topology, ii) multiplexing multiple virtual edge nodes onto a single physical machine, and iii) introducing synthetic background traffic to dynamically change network characteristics and to inject faults.
- We illustrate the generality of our approach through evaluation of a broad range of distributed services, including peer-to-peer systems, ad hoc wireless networking, replicated web services, and self-organizing overlay networks. For one of our experiments, we use ModelNet to independently reproduce published results of a wide-area evaluation of CFS [6].

Our intent is for the research community to adopt large-scale network emulation as a basic methodology for research in experimental Internet systems.

The rest of this paper is organized as follows. The next section describes the ModelNet architecture. Section 3 discusses our implementation and an evaluation of the system's baseline accuracy and scalability. Section 4 then discusses techniques to support accuracy versus scalability tradeoffs in large-scale system evaluation. In Section 5, we demonstrate the generality of our approach by using ModelNet to evaluate a broad range of networked services. Section 6 compares our work to related efforts and Section 7 presents our conclusions.

2 ModelNet Architecture

Figure 1 illustrates the physical network architecture supporting ModelNet. Users execute a configurable number of instances of target applications on *Edge Nodes* within a dedicated server cluster. Each instance is a *virtual edge node* (VN) with a unique IP address and corresponding location in the emulated topology. Edge nodes can run any OS and IP network stack and may execute unmodified application binaries. To the edge nodes, an accurate emulation is indistinguishable from direct execution on the target network. ModelNet emulation runs in *real time*, so packets traverse the emulated network with the same rates, delays, and losses as the real network. We

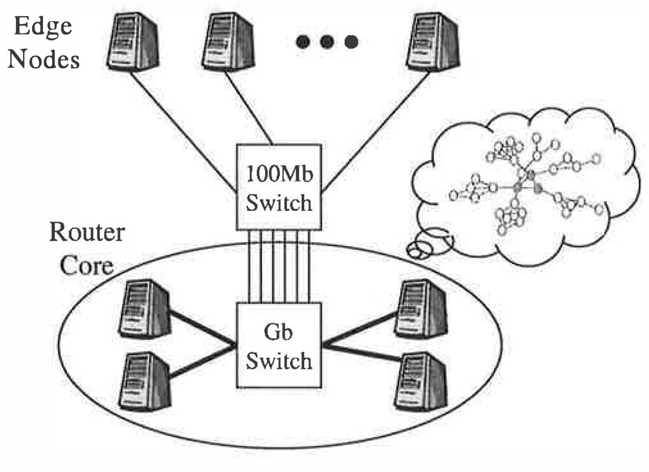


Figure 1: ModelNet.

use standard administrative tools to configure edge nodes to route their network traffic through a separate set of servers acting as *Core Routers*. The core nodes are equipped with large memories and modified FreeBSD kernels that enable them to emulate the behavior of a configured *target network* under the offered traffic load.

A key difference between ModelNet and earlier efforts is that it targets the emulation of entire large-scale topologies. Thus, ModelNet captures the effects of congestion and cross traffic on end-to-end application behavior. To achieve this, the core routes traffic through a network of emulated links or *pipes*, each with an associated packet queue and queueing discipline. Packets move through the pipes and queues by reference; a core node never copies packet data. Packets enter the queues as they arrive from VNs or exit upstream pipes, and drain through the pipes according to the pipe's specified bandwidth, latency, and loss rates. Each queue buffers a specified maximum number of packets; overflows result in packet drops. When a packet exits the link network, the core transmits the packet to the edge node hosting the destination VN.

2.1 ModelNet Phases

ModelNet runs in five phases as shown in Figure 2. The first phase, *Create*, generates a network topology, a graph whose edges represent network links and whose nodes represent clients, stubs, or transits (borrowing terminology from [3]). Sources of target topologies include Internet traces (e.g., from Caida), BGP dumps, and synthetic topology gener-

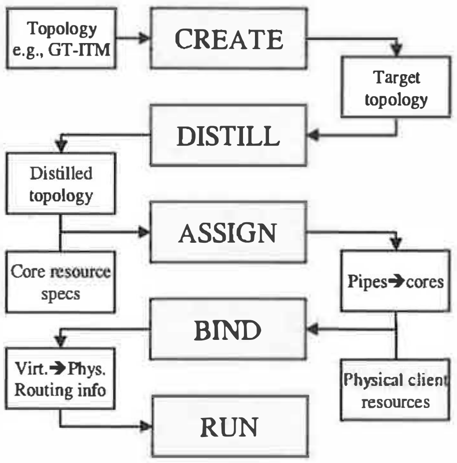


Figure 2: The phases of the ModelNet architecture.

ators [3, 4, 12]. ModelNet includes filters to convert all of these formats to GML (graph modeling language). Users may annotate the GML graph with attributes not provided by its source, such as packet loss rates, failure distributions, etc.

The *Distillation* phase transforms the GML graph to a pipe topology that models the target network. As an option, distillation may simplify the network, trading accuracy for reduced emulation cost. Section 4.1 discusses distillation and other techniques to balance accuracy and scalability in ModelNet.

The *Assignment* phase maps pieces of the distilled topology to ModelNet core nodes, partitioning the pipe graph to distribute emulation load across the core nodes. Note that the ideal assignment of pipes to cores depends upon routing, link properties, and traffic load offered by the application, an NP-complete problem. Currently, we use a simple greedy k-clusters assignment: for k nodes in the core set, randomly select k nodes in the distilled topology and greedily select links from the current connected component in a round-robin fashion. We are investigating approximations for dynamically reassigning pipes to cores to minimize bandwidth demands across the core based on evolving communication patterns.

The *Binding* phase assigns VNs to edge nodes and configures them to execute applications. ModelNet multiplexes multiple VNs onto each physical edge node, then binds each physical node to a single core. The binding phase automatically generates a set of configuration scripts for each node hosting the emulation. For core routers, the scripts install sets of pipes in the distilled topology and routing tables with shortest-path routes between all pairs of VNs. The scripts further configure edge nodes with appropriate IP addresses for each VN.

The final *Run* phase executes target application code on the edge nodes. We have developed simple scripts to automate this process, so that a single command is sufficient to execute thousands of instances of a particular application across a cluster. A key detail is that VNs running application instances must `bind` IP endpoints to their assigned IP addresses in the emulated network rather than the default IP address of the host edge node. This is necessary to mark outgoing packets with the correct source and destination addresses, so they are routed through the core and to their final destination correctly. ModelNet includes a dynamic library to interpose wrappers around the socket-related calls that require this binding step. Most modern operating systems provide a mechanism to preload libraries that intercept system calls while leaving the default variants accessible to the wrapper. In Linux, the affected system calls include `bind`, `connect`, `sendto`, `recvfrom`, `sendmsg`, `recvmsg`, and name resolution calls (e.g., `gethostbyname`, `gethostbyname2`, `gethostname`, `uname` and `gethostbyaddr`).

2.2 Inside the Core

During the *Binding* phase, ModelNet pre-computes shortest-path routes among all pairs of VNs in the distilled topology, and installs them in a routing matrix in each core node. Each route consists of an ordered list of pipes that need to be traversed to deliver a packet from source to destination. This straightforward design allows fast indexing and scales to 10,000 VNs, but the routing tables consume $O(n^2)$ space.

This scheme can be extended to scale to larger target networks. For common Internet-like topologies that cluster VNs on stub domains, we could spread lookups among hierarchical but smaller tables, trading less storage for a slight increase in lookup cost. Another alternative is to use a hash-based cache of routes for active flows (of size $O(n \lg n)$). In the rare case of a cache miss, the route may be fetched from an external cache or computed on the fly from an internal representation of the topology using Dijkstra's shortest path algorithm (an $O(n \lg n)$ operation).

Figure 3 illustrates the packet processing steps in ModelNet cores. First an IP firewall (ipfw) rule intercepts packets entering the emulated network based

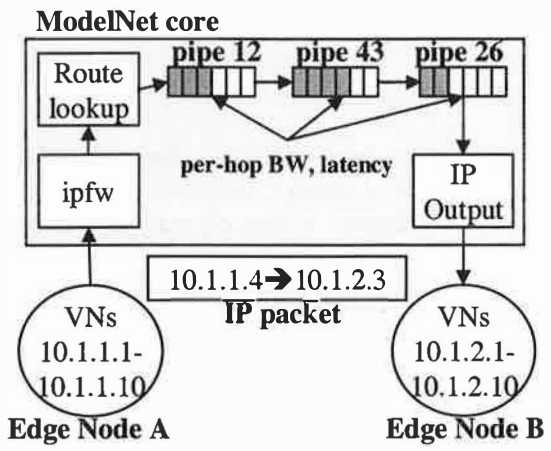


Figure 3: A packet traveling from one edge node to another through ModelNet.

on IP address: all VNs bind to IP addresses of the form *10.0.0.0/8*. On a match, control is transferred to the ModelNet kernel module, which first looks up the route for the given source and destination. This returns the set of pipes that are traversed in the emulated topology. ModelNet creates a descriptor referencing the buffered packet and schedules this descriptor on the appropriate pipes. Because these pipes are shared among all simultaneous flows and each has a fixed (specifiable) amount of queueing, ModelNet is able to emulate the effects of congestion and packet drops according to application-specific communication patterns.

Packet scheduling in ModelNet uses a heap of pipes sorted by earliest deadline, where each pipe's deadline is defined to be the exit time for the first packet in its queue. The ModelNet scheduler executes once every clock tick (currently configured at 10Khz) and runs at the kernel's highest priority level. The scheduler traverses the heap for all pipe deadlines that are later than the current time. The descriptor for the packets at the head of the queue for each of these pipes is removed and either: i) moved to the tail of the queue for the next pipe on the packet's path from source to destination, or ii) the packet itself is scheduled for delivery (using the kernels `ip_output` routine) in the case where the packet has reached its destination. The scheduler then calculates the new deadline for all pipes that had packets dequeued during the current clock tick and reinserts these pipes into the pipe heap according to this new deadline.

Achieving emulation accuracy requires careful coordination of kernel components. In steady state, a ModelNet core performs two principal tasks. First, it processes hardware interrupts to retrieve packets from the network interface. Second, the ModelNet scheduler wakes up periodically to move packets from pipe to pipe or from pipe to final destination. The second operation operates at a strictly higher kernel priority than the first. Thus, under load, ModelNet will preferentially emulate the delay for packets that have already entered the core rather than service packets waiting to enter the core through hardware interrupts. This means that core CPU saturation results in dropped packets (at the physical level) rather than inaccurate emulation. Hence, the relative accuracy of a ModelNet run is proportional to the number of physical packets dropped (note the distinction between physical drops and emulated "virtual" drops in the core).

We base our core implementation on dummynet [18], with extensions to improve accuracy and to support multi-hop and multi-core emulation. When a packet arrives at a pipe queue, the emulation computes the time to dequeue the packet and enter the pipe itself—if it is not dropped due to a congestion-related queue overflow, randomized loss, or a RED policy (each pipe is FIFO by default). We calculate this time based on the size of the packet, the size of all earlier packets queued waiting to enter the pipe, and the bandwidth of the pipe. As described above, the clock interrupt handler checks for dequeued packets once every system tick. A packet enters a pipe by transferring to the pipe's *delay line* queue, where it waits until it exits the pipe according to the pipe's specified latency; the link's delay-line queue holds its bandwidth-delay product if the link is fully utilized.

This process repeats for every pipe in the path from source to destination. When the packet exits its last pipe, `ip_output` forwards it to its destination edge node, where the host operating system delivers it to the VN process bound to the destination IP address.

For multi-core configurations, the next pipe in a route may be owned by a different core node. In this case, the core node tunnels the packet descriptor to the owning node, which is determined by a table lookup in a pipe ownership directory (POD) created during the Binding phase. Note that link emulation does not require access to the packet contents itself, so ModelNet can reduce the physical bandwidth requirements in the core by leaving the packet contents buffered on the entry core node and forwarding it directly to the destination edge node when it exits the emulated network [22].

2.3 Discussion

We now briefly discuss a number of outstanding issues with the current ModelNet architecture. Since we do not perform resource isolation among competing VN's running on the edges, it is possible for a single VN to transmit UDP packets as fast as allowed by the hardware configuration (100 Mbps in our setup). ModelNet will drop the appropriate portion of these packets according to the specified first-hop characteristics of emulated pipes in the topology. However, since UDP flows do not respond to congestion signals (dropped packets), they will continue sending at the same rate, preventing other well-behaving TCP flows originating from the same physical host to obtain their proper share of emulated bandwidth. A number of solutions exist to this problem, such as running individual VN's within a virtual machine or running traffic shapers (ModelNet itself could be applied recursively in this manner) on the edge hosts. For portability, simplicity, and our desire to focus on congestion-friendly distributed services, we choose to minimize required modifications to edge nodes.

There are a number of scalability issues with ModelNet. First, the traffic traversing the ModelNet core is limited by the cluster's physical internal bandwidth. For current commodity network switches, this means that application bisection bandwidth is limited to roughly 10 Gb/s—assuming 10 ModelNet cores on a switched gigabit network, and that individual pipes in the emulated topology are limited to the bandwidth of a single host, or 1 Gb/s. We believe this level of bandwidth to be sufficient for a wide variety of interesting distributed services. Next, Modelnet must buffer up to the full bandwidth-delay product of the target network. Fortunately, this memory requirement is manageable for our target environment. For example, a core node requires only 250 MB of packet buffer space to carry flows at an aggregate bandwidth of 10 Gb/s (currently beyond the capacity of a single core node) with 200 ms average round-trip latency.

Finally, we currently assume the presence of a "perfect" routing protocol that calculates the shortest path between all pairs of hosts. Upon an individual node or link failure, we similarly assume that the routing protocol is able to instantaneously discover the resulting shortest paths. We are currently in the process of implementing various routing protocols within the ModelNet core. The idea here is to emulate the propagation and processing of routing protocol packets within a ModelNet routing module without involving edge nodes. By leveraging our existing packet emulation environment, we are able to capture the latency and communication overhead associated with routing protocol code while leaving the edge hosts unmodified.

3 Implementation and Evaluation

We have completed an implementation of the entire ModelNet architecture described above. Here, we describe the novel aspects of the resulting implementation, focusing on ModelNet's performance and scalability. Unless otherwise noted, all experiments in this paper run on the following configuration, as depicted at a high level in Figure 1. The ModelNet core routers are 1.4Ghz Pentium IIIs with 1GB of main memory running FreeBSD-4.5-STABLE with a separate kernel module running our extensions. Each core router possesses a 3Com 3c966-T NIC based on the Broadcom 5700 chipset and is connected to a 12-port 3Com 4900 gigabit switch. Edge nodes are 1Ghz Pentium IIIs with 256MB of main memory running Linux version 2.4.17 (though we have also conducted experiments with Solaris 2.8 and FreeBSD-4.x). Unless otherwise noted, these machines are connected via 100Mb/s Ethernet interface to multiple 24-way 3Com 100 Mb/s switches with copper gigabit aggregating uplinks to the 3Com 4900 switch connecting the ModelNet core.

3.1 Baseline Accuracy

The ability to accurately emulate target packet characteristics on a hop-by-hop basis is critical to ModelNet's success. Thus, a major component of our undertaking was to ensure that, under load, packets are subject to appropriate end-to-end delays, as determined by network topology and prevailing communication patterns. We developed a kernel logging package to track the performance and accuracy of ModelNet. The advantage of this approach is that information can be efficiently buffered and stored offline for later analysis. Using such profiling, we were able to determine the expected and actual delay on a per-packet basis. We found that by running the ModelNet scheduler at the kernel's highest priority level, each packet-hop is accurately emulated to within the granularity of the hardware timer (currently 100 μs) up to and including 100% CPU utilization. Recall that packets are physical dropped on the core's network interface under overload. If a packet takes 10 hops from source to destination, this corresponds to a worst-case error of 1 ms, which we consider acceptable for our target wide-area experiments. We are in the process of implementing packet debt handling within the ModelNet scheduler, where the scheduler maintains the total emulation error and attempts to correct for it in subsequent hops. With this optimization, we believe that per-packet emulation accuracy can be reduced to 100 μs in all cases.

ModelNet delivers packets to their destination within 1 ms of the target end-to-end value that accounts for propagation delay, queueing delay, and hop-by-hop link bandwidth. The system maintains this accuracy up to and including 100% CPU utilization. Given that the principal focus of our work is large-scale wide-area network services, we believe that this level of accuracy is satisfactory.

3.2 ModelNet Capacity

Our first experiment quantifies the capacity of ModelNet as a function of offered load, measured in packets per second (like most "routers", ModelNet overhead is largely independent of packet size save the relatively modest memcpy overhead) and as a function of emulated hops that the packet must traverse. We vary from 1-5 the number of edge nodes with 1 Gb/s Ethernet connections communicating through a single core with a 1 Gb/s link. Each edge node hosts up to 24 netperf senders (24 VNs) communicating over TCP. We spread the receivers across 5 edge nodes, each also hosting up to 24 netperf receivers. The topology directly connects each sender with a receiver over a configurable number of 10 Mb/s pipes with an end-to-end latency of 10 ms. By changing the number of pipes in the path, we vary the total amount of work ModelNet must perform to emulate end-to-end flow characteristics.

Figure 4 plots the capacity in packets/sec of the single ModelNet core as a function of the number of simultaneous flows competing for up to 10 Mb/s of bandwidth each. There are five curves, each corresponding to a different number of pipes that each packet must traverse from source to destination. For the baseline case of 1 hop, Figure 4 shows that the performance of the core scales linearly with offered load up to 96 simultaneous flows (4 edge nodes). At saturation with 1 hop, ModelNet accurately emulates approximately 120,000 packets/sec. At this point, the ModelNet CPU is only 50% utilized, with the bottleneck being in the network link. 120,000 packets/sec corresponds to an aggregate of 1 Gb/s (aver-

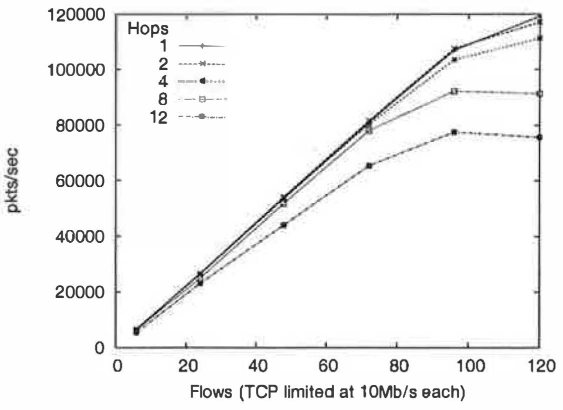


Figure 4: Capacity of a single ModelNet core.

age packet size is 1 KB when accounting for 1 ACK for every two 1500-byte data packets).

Only once we attempt to emulate more than 4 hops per flow does the ModelNet CPU become the bottleneck. For 8 hops per flow, ModelNet accurately forward approximately 90,000 packets/sec. At this point, the machine enters a state where it consumes all available resources to i) pull 90,000 packets/sec from the NIC, ii) emulate the 8-hop characteristics of each packet, and finally iii) forward the packets to their destination. The NIC drops additional packets beyond this point (recall that ModelNet emulation runs at higher priority than NIC interrupt handling). These drops throttle the sending rate of the TCP flows to the maximum 90,000 packets/sec. Overall, we measure a 0.5 μs overhead for each hop in the emulated topology and a fixed per-packet overhead of 8.3 μs for each packet traversing the core (resulting from the standard IP protocol stack). Our hardware configuration can forward approximately 250,000 small packets/sec when not performing ModelNet emulation, indicating significant opportunity for improved system capacity with improving CPU performance.

3.3 Scaling with Additional Cores

The next experiment measures ModelNet's ability to deliver higher throughput with additional communicating core routers. Recall that ModelNet's ability to deliver additional bandwidth with additional cores depends upon the target topology and application communication patterns. As the topology is divided across an increasing number of cores, there is an increasing probability that a packet's path forces it to cross from one core to another before being deliv-

Cross-Core Traffic	Throughput (Kpkt/sec)
0%	462.5
25%	404.5
50%	276.3
75%	219.3
100%	155.8

Table 1: Scalability as a function of communication pattern for 4-core experiment.

ered to its final destination. Each such core crossing negatively impacts overall system scalability.

To quantify this effect, we configure 20 machines as edge nodes, each hosting 56 VNs, evenly split among 28 senders and 28 receivers (for a total of 1120 VNs). For this experiment, the edge machines were each configured with a gigabit link to the ModelNet core to allow us to load the 4-node configuration with fewer resources. We emulate a star topology with all VNs connected to a central point with 10 Mb/s, 5 ms latency pipes. Thus, all paths through the topology consist of 2 hops. We assign one quarter of the VNs to each core. During the experiment all 560 senders use `netperf` to simultaneously transmit TCP streams to 560 unique receivers; there is no contention for the last-hop pipes. We vary the communication pattern so that during any one experiment, x% of flows must cross from one core to the other, leaving $(1-x)$% of the traffic to travel within a single core.

Table 1 summarizes the results of this experiment, showing the maximum system throughput in packets/sec as a function of the percent of cross-core traffic. With 0% cross-core traffic, the system delivers four times the throughput of a single core (compare to the 2-hop results from Figure 4 above) degrading roughly linearly with additional cross-core traffic. For more complicated topologies where, for example, each packet must traverse four different cores, throughput would degrade further. Thus, the ability to scale with additional core nodes depends on application communication characteristics and properly partitioning the topology to minimize the number of inter-core packet crossings.

4 Accuracy versus Scalability Trade-offs

This section outlines several techniques to configure emulations on a continuum balancing accuracy and cost. It is impractical to model every packet and link for a large portion of the Internet on a PC cluster. Rather, the goal of ModelNet is to create a controlled Internet-like execution context with a diversity of link capabilities, rich internal topology, network locality, dynamically varying link status, cross traffic, congestion, and packet loss. The key to emulation at scale is to identify and exploit opportunities to reduce overhead by making the emulation less faithful to the target network in ways that minimally impact behavior for the applications under test. Ideally, the emulation environment would automate these trade-offs to yield the most accurate possible emulation on the available hardware, then report the nature and degree of the inaccuracy back to the researcher. We have taken some initial steps toward this vision in our work with ModelNet.

4.1 Distillation

ModelNet's *Distillation* phase modifies the topology with the goal of reducing the diameter of the emulated network. In a pure *hop-by-hop* emulation, the distilled topology is isomorphic to the target network; ModelNet faithfully emulates every link in the target network. This approach captures all congestion and contention effects in the topology but also incurs the highest per-packet overhead. At the other extreme, *end-to-end*, distillation removes all interior nodes in the network, leaving a full mesh with $O(n^2)$ links interconnecting the n VNs. The distiller generates the mesh by collapsing each path into a single pipe p: the bandwidth of p is the minimum bandwidth of any link in the corresponding path, its latency is the sum of the link latencies along the path, and its reliability $(1 - lossrate)$ is the product of the link reliabilities along the path. This approach yields the lowest per-packet emulation overhead—since each packet traverses a single hop—and it accurately reflects the raw network latency, bandwidth, and base loss rate between each pair of VNs. However, it does not emulate link contention among competing flows except between each pair of VNs.

ModelNet provides a *walk-in* "knob" to select an arbitrary balance along the continuum between these extremes by "preserving" the first *walk-in* links from the edges. A breadth-first traversal generates successive *frontier sets*. The first frontier set is the set of all VNs; the members of the $i+1$ frontier set are the nodes that are one hop from a member of the ith frontier set, but are not members of any preceding frontier set. The *interior* consists of all nodes that are not members of the first *walk-in* frontier sets.

The distiller replaces the interior links of the target topology with a full mesh interconnecting the interior nodes, as in the end-to-end approach. This approach reduces emulation cost relative to hop-by-hop because each packet traverses at most *(2*walk-in)+1* pipes, rather than the network diameter. A *last-mile* emulation (*walk-in = 1*) preserves the first and last hop of each path.

Distilled *walk-in* emulations do not model contention in the interior, but this is a useful tradeoff assuming that today's Internet core is well-provisioned and that bandwidth is constrained primarily near the edges of the network at peering points and over the last hop/mile. To model under-provisioned cores, ModelNet extends the *walk-in* algorithm above to preserve the inner core of breadth *walk-out*. This extends the *walk-in* algorithm above to generate successive frontier sets until a frontier of size one or zero is found, representing the topological "center" of the target topology. Suppose that this is frontier set *c*. The interior corresponding to breadth *walk-out* is the union of frontier sets *c - walk-out* through *c*. Distillation preserves links interconnecting the interior, while collapsing paths between the walk-in and walk-out frontiers as described above.

To illustrate distillation, we ran a simple experiment with a ring topology. The ring has 20 routers interconnected at 20 Mb/s; each ring router has 20 VNs directly connected to it by individual 2 Mb/s links. The VNs are evenly partitioned into *generator* and *receiver* sets of size 200. Each generator transmits a TCP stream to a random receiver. This topology then has 419 pipes shared among the 400 VNs. The end-to-end distillation contains 79,800 pipes, one for each VN pair, each with a bandwidth of 2 Mb/s. The last-mile distillation preserves the 400 edge links to the VNs, and maps the ring itself to a fully connected mesh of 190 links. The last-hop configuration can handle significantly more packets on a given emulation platform, since each packet traverses 3 hops rather than the average case of 7 in the original ring topology.

Figure 5 plots a CDF for the distribution of bandwidth achieved by the 200 flows in the various emulations, and in full hop-by-hop simulations using the ns2 simulator with both 20 Mb/s and 80 Mb/s rings. The hop-by-hop emulation shows a roughly even distribution of flow bandwidths, matching the 20 Mb/s ns2 results. Here, each 20 Mb/s transit link has an offered load of 27.5 Mb/s, constraining the bandwidth of flows passing through the ring. In

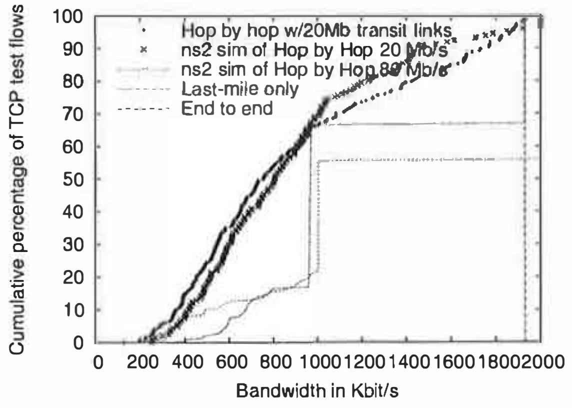


Figure 5: The effects of distillation on distribution of bandwidth in a ring topology.

the end-to-end emulation, which does not model contention in the ring, all flows achieve their full 2 Mb/s of bandwidth. The last-mile emulation models congestion only for the 64% of flows that share the same receiver; the bandwidths for these flows are 1 Mb/s or less, depending on the number of flows sharing a given receiver. The other 36% of flows all achieve 2Mb/s. The last-mile emulation is qualitatively similar to the ns2 results with an 80 Mb/s ring, since in this case the ring is adequately provisioned and the only contention is on the last hop to the receiver. The emulation results closely match the exhaustively verified ns2 simulation environment, improving our confidence in the accuracy of our implementation.

4.2 VN Multiplexing

The mapping of VNs to physical edge nodes also affects emulation accuracy and scalability. Higher degrees of multiplexing enable larger-scale emulations, but may introduce inaccuracies due to context switch overhead, scheduling behavior, and resource contention on the edge nodes. The maximum degree of safe multiplexing is application-dependent. For many interesting application scenarios (e.g., Web load generators), the end node CPU is not fully utilized and higher multiplexing degrees are allowable. The multiplexing degree is lower for resource-intensive VNs, such as Web servers.

The accuracy cost of VN multiplexing also varies with the concurrency model. In general, each VN runs as one or more processes over the edge node host OS; all sockets open in those processes bind to the VN's assigned IP address. Mapping the sockets of a single process onto multiple VNs may allow

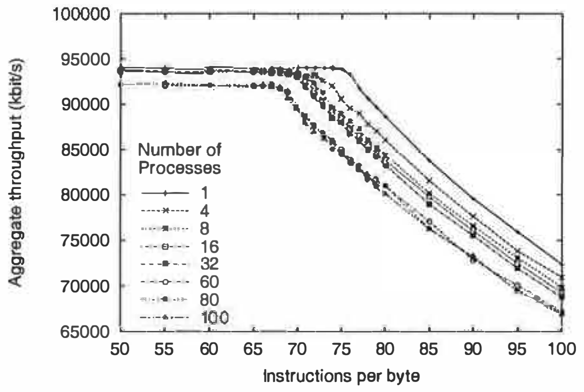


Figure 6: Effects of multiplexing processes on an edge node.

higher VN multiplexing degrees with less likelihood of saturating the edge node if the process manages its concurrency efficiently using threads or an event-driven model. ModelNet users can achieve this per-socket VN mapping in a straightforward manner using a variant of the socket interposition library that maps each open socket to a different VN.

To illustrate the accuracy tradeoff of VN multiplexing, we ran an experiment to show its effect on VNs running modified netperf/netserver processes that exchange 1500-byte UDP packets with a configurable amount of computation per packet. The experiment uses three physical nodes: one for netperf sources, one for netserver sinks, and one for the ModelNet core. We ran experiments with varying multiplexing degrees, varying the emulated topology so that each of *nprog* netperf/netserver pairs is configured with 1/*nprog* of the physical 100 Mb/s link.

Figure 6 plots aggregate bandwidth across all processes as a function of the number of instructions *delay* (per byte) after each packet transmission, for a multiplexing degree *nprog* ranging from 1 to 100. The actual delay in instructions between packet transmissions on the x-axis should be multiplied by the packet size (1500 bytes). We expect context switch overhead to consume an increasing fraction of system resources as we increase *nprog*. With zero *delay*, the processes deliver an aggregate of approximately 95 Mb/s. With *nprog* = 1, there is no loss of throughput up to a delay of 76 instructions/byte. The theoretical maximum is 1Ghz*8/100Mb/s = 80 instructions per byte (assuming a CPI of 1.0). With *nprog* = 2, the allowable total computation per-byte degrades to 73 instructions due to context switching overheads; with *nprog* = 100 it falls to 65 instructions per transmitted byte.

We plan to use such a benchmarking tool to automate the process of determining the computation versus communication ratio of individual applications (with the help of the Pentium's cycle and instruction counters). Such an evaluation will assist ModelNet users to determine the accuracy tradeoff at various multiplexing degrees.

4.3 Changing Network Characteristics

An important goal of ModelNet is to enable researchers to evaluate adaptive Internet systems by subjecting them to competing traffic and observing their responses to the resulting variations in network performance. ModelNet users may do this directly by incorporating generators for competing traffic with specified properties (TCP, constant bit rate, etc.) into the VN application mix. While this is the most accurate way to emulate "background" cross traffic, it consumes resources at the edge nodes as well as ModelNet emulation bandwidth at the core. Another option is to modify the core kernels to insert dummy packets into the pipes according to a specified pattern. However, this approach also consumes significant core resources to generate the cross traffic and propagate it through the pipe network.

ModelNet allows users to inject cross traffic by modifying pipe parameters dynamically as the emulation progresses. The cross traffic at each point in time is specified as a matrix indicating communication bandwidth demand between each VN pair (i, j). These matrices may be generated from a synthetic background traffic pattern or probability distribution, or fetched from a stored set of "snapshot" profiles. An offline tool propagates the matrix values through the routing tables to determine the impact of the specified cross-traffic signals on each pipe in the distilled topology. During the emulation, a configuration script periodically installs derived pipe parameter settings into the ModelNet core nodes. The new settings represent cross traffic effects by increasing pipe latency to capture queueing delays, reducing pipe bandwidth to capture the higher link utilization, and reducing the queue size bound to model the impact on the steady-state queue length. Thus, a flow competing with synthetic cross traffic sees a reduced ability to handle packet bursts without loss, as well as increased latency and lower available bandwidth. We derive the new settings from a simple analytical queueing model for each impacted link.

This approach incurs low overhead and scales independently of both the cross traffic rate and the actual traffic rate. The drawback is that it does not capture all the details of Internet packet dynamics (slow start, bursty traffic, etc.). In particular, synthetic cross traffic flows are not responsive to congestion in our current approach; thus they introduce an emulation error that grows with the link utilization level.

ModelNet also adjusts pipe parameter settings to emulate other dynamic network changes, e.g., fault injection. For example, the user can direct ModelNet to change the bandwidth, delay, and loss rate of a set of links according to a specified probability distribution every x seconds. For node or link failures, the configuration script also updates the system routing tables. Currently, this is done by recalculating all-pairs shortest paths; we are currently investigating techniques for emulating realistic routing protocols within ModelNet. In this way, users can observe how their system responds to pre-specified stimuli, for example, network partitions, sudden improvements in available bandwidth, sudden increase in loss rate across a backbone link, etc. In particular, random stress tests are useful because it is often just as important to identify conditions under which services will fail than it is to demonstrate how well they behave in the common case.

5 Case Studies

In this section, we demonstrate the utility and generality of our approach by evaluating four sample distributed services on ModelNet. Beyond the tests below, the largest single experiment completed in our environment successfully evaluated system evolution and connectivity of a 10,000 node network of unmodified gnutella clients by mapping 100 VNs to each of 100 edge nodes in our cluster. To further explore the generality of our approach, we have also implemented extensions to our architecture to support emulation of ad hoc wireless environments. These changes involve supporting the broadcast nature of wireless communication (packet transmission consumes bandwidth at all nodes within communication range of the sender) and node mobility (topology change is the rule rather rather than the exception). While complete, we omit a detailed evaluation of this last case study for brevity.

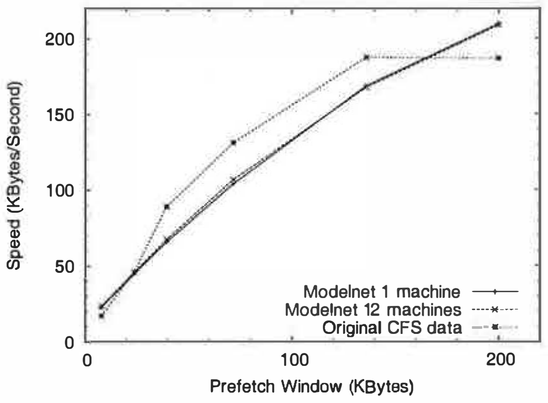


Figure 7: Download speed as a function of prefetch window for CFS/RON and CFS/ModelNet.

5.1 CFS

As a first demonstration of the utility of ModelNet, we set out to reproduce the published experimental results of another group's research in large-scale distributed computing. We chose the CFS implementation [6] running on top of the RON testbed [1]. CFS is an archival storage system layered on top of Chord [20], a fully distributed hash table. A CFS prototype was evaluated on 12 nodes spread across the Internet. We chose CFS running on RON because of their exemplary experimental practice: The CFS code is publicly available and the network characteristics (bandwidth, latency, loss rate) among all pairs of RON nodes are published.

We converted the publicly available RON interconnectivity characteristics into a ModelNet topology. The only limitation we encountered was that the authors did not record exactly which 12 of the 15 RON nodes hosted the reported experimental results. We tried a number of permutations of potential participants, but were not exhaustive because, as shown below, we were able to closely reproduce the CFS results in all cases.

Specifically, we reproduce the experimental results from Figures 6, 7, and 8 of the CFS paper [6]. CFS Figures 6 and 7 are two views of how varying the Chord prefetch size changes the download bandwidth for retrieving 1MB of data striped across all participating CFS nodes. CFS Figure 8 shows the bandwidth distribution for a series of pure TCP transfers between the RON nodes and is presented as a comparison between the transfer speeds delivered by CFS relative to TCP. We were able to extract the

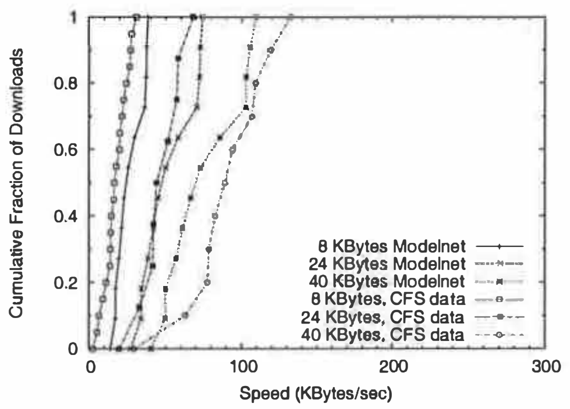


Figure 8: CDF of download speed for various prefetch windows for CFS/RON and CFS/ModelNet.

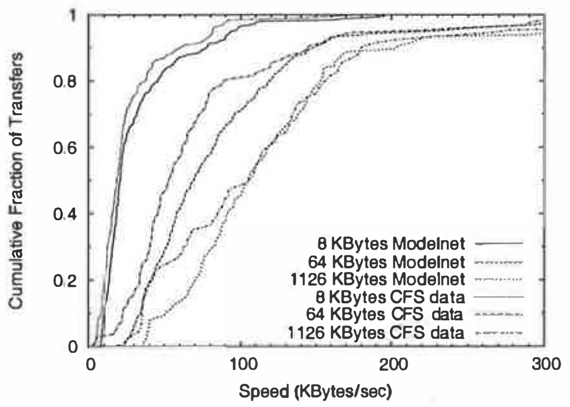


Figure 9: TCP transfer of 8, 64, and 1164KB files using TCP running on both the RON testbed and ModelNet.

raw data for these experiments from information contained within the encapsulated postscript of the figures in the publicly available postscript version of the CFS paper. We plot these curves along with our own experimental results of the CFS code running on ModelNet in Figures 7, 8, and 9.

We replicate the Chord prefetch experiment (Figures 7 and 8) in two ways. The "Modelnet 12 node" curve shows the results of running the experiment on 12 individual edge nodes, each with a single VN representing a RON node and running CFS code. Next, to further demonstrate ModelNet's ability to accurately multiplex multiple logical nodes onto a single physical node, we run 12 instances (VNs) of CFS on a single machine. The "ModelNet 1 node" curve in Figure 7 shows the results of this experiment. Figure 8 depicts the Chord prefetch results in more detail.

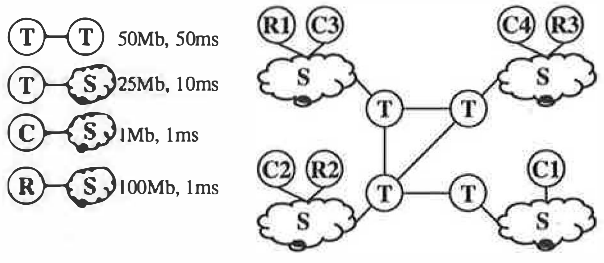


Figure 10: Sample network topology for investigating the effects of replication on client-perceived latency.

A final experiment in CFS measures basic TCP bandwidth between all RON nodes by transferring files of 8, 64 and 1164 KB. Figure 9 shows the results for each transfer size sorted by achieved throughput for both our ModelNet experiments and the CFS wide-area experiments. We used 12 physical edge nodes for this experiment.

Overall, the results of CFS/ModelNet experiments closely match the results for CFS/RON in all cases. Several potential sources of error may explain the discrepancies: i) the CFS experiments were run at a different time than the RON network characteristics were measured, ii) resource contention in the "middle" of the network were not captured by the end-to-end network characterization available to us, and iii) different hardware and operating systems were used to carry out the wide-area versus ModelNet experiments. Overall, however, we are quite satisfied with how closely we were able to reproduce these results. Running experiments on ModelNet is also a lightweight operation relative to coordinating experiments across multiple wide-area nodes: One experienced programmer was able to reproduce all the CFS experiments on ModelNet in one day.

5.2 Replicated Web Services

Recently, there has been increasing interest in wide-area replication of Internet services to improve overall performance and reliability. A principal motivation for our work is to develop an environment to support the study of replica placement and request routing policies under realistic wide-area conditions. While such a comprehensive study is beyond the scope of this paper, we present the results of some initial tests to demonstrate the suitability of ModelNet for evaluating replicated network services.

The goal of our experiment is to demonstrate that ModelNet support for realistic Internet topologies

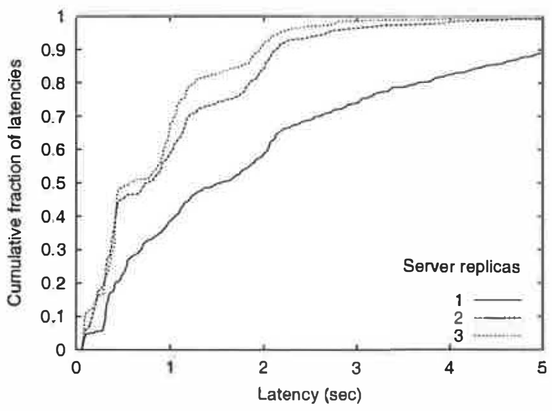


Figure 11: CDF of client-perceived latency as a function of number of replicas.

and emulation of contention for shared pipes supports complex experiments involving replica placement and request routing policies. We create a modified 320-node transit-stub [3] topology, as depicted in Figure 10. Clouds labeled "S" in the figure represent stub domains with more complex internal connectivity. These stub-stub links are set with 10 Mb/s bandwidth, and 5 ms latency. Additional link characteristics are set as described in the figure. For our experiments, we use locally developed software to play back in real time 2.5 minutes of a larger trace to IBM's main web site (`www.ibm.com`) from February 2001 [5]. Load varies from 60-100 requests/sec during this period. We use eight machines for this experiment: a single ModelNet core, 4 machines (hosting clients labeled $C1 - C4$ in Figure 10) simultaneously play back the trace, and up to 3 machines (labeled $R1 - R3$ in Figure 10) host Apache (version 1.3.24-3) servers. We map 30 VNs in the same stub domain to each of $C1 - C4$. Each VN has its own dedicated 1 Mb/s, 1 ms link to the rest of the stub domain. We run three separate experiments. In the first, all clients $C1 - C4$ make requests to $R1$. Server CPU utilization was 10% even when all requests are directed to this single server, indicating that the network rather than host CPU was the bottleneck in our experiments. In the second experiment, we manually configure clients at $C1$ and $C2$ to direct their requests to $R2$ (the remaining clients still go to $R1$). In the final experiment, we configure $C4$ to direct its requests to $R3$.

Figure 11 plots the cumulative distribution of all client latencies for the three different experiments described above. We see that with 1 server, client latencies are relatively large, with 10% taking longer than 5 seconds to complete. Instrumentation shows that this results from contention on the transit-to-transit links in the topology. An additional replica greatly improves client latency by largely eliminating such contention. A third replica provides only marginal benefit in this case. Note that these results would not be possible without ModelNet's ability to accurately emulate contention on interior links in the topology. Further, these simple results depict an upper bound on the benefits of wide-area replication because we assume a perfect request routing mechanism and static network conditions. Of course, a more comprehensive experiment must support dynamic request routing decisions (e.g., leveraging DNS in a content distribution network) and dynamically changing network conditions.

5.3 Adaptive Overlays

We use ACDC [9], an application-layer overlay system that dynamically adapts to changing network conditions, to demonstrate ModelNet's ability to subject systems to dynamically changing network conditions. ACDC attempts to build the lowest-cost overlay distribution tree that meets target levels of end-to-end delay. Cost and delay are two independent and dynamically changing metrics assigned to links in the underlying IP network. Nodes in the overlay use a distributed algorithm to locate parents that deliver better cost, better delay, or both. A key goal is scalability: no node maintains more than $O(\lg n)$ state or probes more than $O(\lg n)$ peers.

Full details of the algorithm and an ns2-based evaluation are available elsewhere [9]. We wrote ACDC to a locally developed compatibility layer that allows the same code to run both in ns2 and under live deployment. Thus, we are able to reproduce the results of an experiment running under both ns2 and ModelNet. We begin with a 600-node GT-ITM transit-stub topology and randomly choose 120 nodes to participate in the ACDC overlay. We assign transit-transit links a bandwidth of 155 Mb/s and a cost of 20-40, transit-stub links 45 Mb/s with a cost of 10-20 and stub-stub links 100 Mb/s with a cost of 1-5. GT-ITM determines propagation delay based on relative location in the randomly generated graph. We run ACDC using a single core and 10 edge nodes, each hosting 12 VNs.

Nodes initially join at a random point in the overlay and self-organize first to meet application-specified delay targets (1500 ms in this experiment) and then to reduce cost while still maintaining delay. Fig-

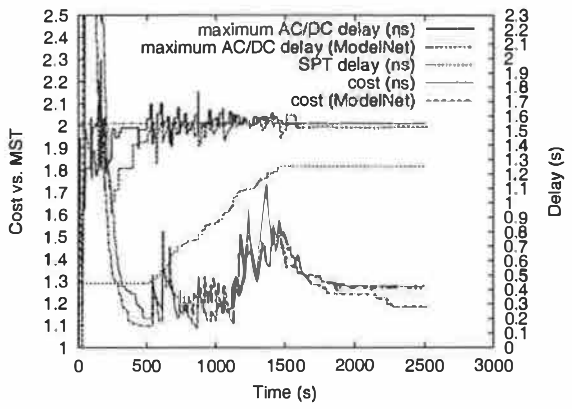


Figure 12: ACDC cost and delay running on ModelNet.

ure 12 plots as a function of time: i) the cost of the overlay relative to a minimum cost spanning tree (calculated offline) on the left y-axis, and ii) the worst-case delay of the overlay on the right y-axis. The graph also depicts the delay through the shortest path tree (again calculated offline) connecting the 120 participants. The closer the SPT delay is to the 1500ms target, the more difficult it is to achieve the performance goal. After allowing the overlay to stabilize for 500 seconds, ModelNet increases the delay on 25% randomly chosen IP links by between 0-25% of the original delay every 25 seconds. The figure shows ACDC's ability to self-organize to maintain target performance levels, sometimes sacrificing cost to do so. At $t = 1500$, network conditions subside and ACDC once again focuses on reducing cost. For comparison, Figure 12 also plots the results of an identical experiment under ns2. Overall, the results for ACDC/ns2 closely match those for ACDC/ModelNet.

6 Related Work

Many previous efforts investigate the use of emulation in support of their research [2, 11, 7, 14, 23]. Relative to ModelNet, these efforts largely focus on specific, static, and relatively small-scale systems. ModelNet, on the other hand, supports a scalable and flexible emulation environment useful for a broad range of research efforts and further performs full hop-by-hop network emulation, allowing it to capture the effects of contention and bursts in the middle of the network. Perhaps most closely related to our effort is Netbed (an outgrowth of Emulab) [21]. This tool allows users to configure a subset of network resources for isolated distributed systems and networking experiments. The testbed provides an integrated environment that allows users to set up target operating systems and network configurations. Relative to Emulab, we focus on scalable emulation of large-scale network characteristics. Overall, we plan to integrate our effort into this system in the future.

A number of collaborative efforts provide nodes across the wide-area to support "live" experimentation with new protocols and services. Examples include Access, CAIRN, PlanetLab [16], Netbed, RON [1] and the X-Bone. While these testbeds are extremely valuable, they are also typically limited to a few dozen sites and do not support controlled, large-scale, and reproducible experiments. We view ModelNet as complementary to these very necessary efforts.

One recent effort [13] uses emulation to evaluate the effects of wide-area network conditions on web server performance. They advocate emulating network characteristics at end hosts rather than in a dedicated core for improved scalability. However, this approach cannot capture the congestion of multiple flows on a single pipe, requires appropriate emulation software on all edge nodes, and must share each host CPU between the tasks of emulation and executing the target application.

7 Conclusions

Ideally, an environment for evaluating large-scale distributed services should support: i) unmodified applications, ii) reproducible results, iii) experimentation under a broad range of network topologies and dynamically changing network characteristics, and iv) large-scale experiments with thousands of participating nodes and gigabits of aggregate cross traffic. In this paper, we present the design and evaluation of ModelNet, a large-scale network emulation environment designed to meet these goals. ModelNet maps a user-specified topology to a set of core routers that accurately emulate on a per-packet basis the characteristics of that topology, including per-link bandwidth, latency, loss rates, congestion, and queueing. ModelNet then multiplexes unmodified applications across a set of edge nodes configured to route all their packets through the ModelNet core.

Of course, no cluster can capture the full scale and complexity of the Internet. Thus, a significant contribution of this work is an evaluation of a set of scalable techniques for approximating Internet conditions. In addition to a detailed architectural evaluation, we demonstrate the generality of our approach by presenting our experience with running a broad range of distributed services on ModelNet, including a peer-to-peer file service, an adaptive overlay, a replicated web service, and an ad hoc wireless communication scenario.

References

[1] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of SOSP 2001*, October 2001.

[2] Guarav Banga, Jeffrey Mogul, and Peter Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference*, June 1999.

[3] Ken Calvert, Matt Doar, and Ellen W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, June 1997.

[4] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *Proceedings of ACM SIGMETRICS*, June 2002.

[5] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, October 2001.

[6] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, October 2001.

[7] NIST Internetworking Technology Group. See `http://www.antd.nist.gov`.

[8] Yang hua Chu, Sanjay Rao, and Hui Zhang. A Case For End System Multicast. In *Proceedings of the ACM Sigmetrics 2000 International Conference on Measurement and Modeling of Computer Systems*, June 2000.

[9] Dejan Kostić, Adolfo Rodriguez, and Amin Vahdat. The Best of Both Worlds: Adaptivity in Two-Metric Overlays. Technical Report CS-2002-10, Duke University, May 2002. `http://www.cs.duke.edu/~vahdat/ps/acdc-full.pdf`.

[10] Balachander Krishnamurthy, Craig Wills, and Yin Zhang. On the Use and Performance of Content Distribution Networks. In *SIGCOMM Internet Measurement Workshop*, 2001 November.

[11] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the 1997 International Symposium on Computer Architecture*, June 1997.

[12] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: An Approach to Universal Topology Generation. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, August 2001.

[13] Erich M. Nahum, Marcel Rosu, Srinivasan Seshan, and Jussara Almeida. The Effects of Wide-Area Conditions on WWW Server Performance. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2001.

[14] Brian Noble, M. Satyananarayanan, Giao Nguyen, and Randy Katz. Trace-based Mobile Network Emulation. In *Proceedings of SIGCOMM*, September 1997.

[15] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[16] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of ACM HotNets-I*, October 2002.

[17] Sylvia Ratnasamy, Paul Francis Mark Handley, Richard Karp, and Scott Shenker. A Content Addressable Network. In *Proceedings of SIGCOMM 2001*, August 2001.

[18] Luigi Rizzo. Dummynet and Forward Error Correction. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[19] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Middleware'2001*, November 2001.

[20] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer to Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 SIGCOMM*, August 2001.

[21] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[22] Kenneth G. Yocum and Jeffrey S. Chase. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *Proceedings of the USENIX Technical Conference*, June 2001.

[23] Haifeng Yu and Amin Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

Pastiche: Making Backup Cheap and Easy

Landon P. Cox, Christopher D. Murray, and Brian D. Noble
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122
{lpcox,cdmurray,bnoble}@umich.edu http://mobility.eecs.umich.edu/

Abstract

Backup is cumbersome and expensive. Individual users almost never back up their data, and backup is a significant cost in large organizations. This paper presents *Pastiche*, a simple and inexpensive backup system. Pastiche exploits excess disk capacity to perform peer-to-peer backup with no administrative costs. Each node minimizes storage overhead by selecting peers that share a significant amount of data. It is easy for common installations to find suitable peers, and peers with high overlap can be identified with only hundreds of bytes. Pastiche provides mechanisms for confidentiality, integrity, and detection of failed or malicious peers. A Pastiche prototype suffers only 7.4% overhead for a modified Andrew Benchmark, and restore performance is comparable to cross-machine copy.

1 Introduction

Backup is cumbersome and expensive. Personal machines are backed up rarely, if at all. Internet backup services exist, but are costly. For example, Connected TLM offers individual users backup of up to 4 GB—but covers neither applications nor the operating system—for $15 per month [15]. Machines within an organization can be backed up centrally, but at significant cost. For example, the computer support arm of Michigan's College of Engineering provides backup service of up to 8 GB on a single machine for $30 per month.

The cost and inconvenience of backup are unavoidable, and often prohibitive. Small-scale solutions require significant administrative efforts. Large-scale solutions require aggregation of substantial demand to justify the capital costs of a large, centralized repository.

There is increasing recognition that disks better serve the needs of near-line archival storage. The purchasing cost of disk subsystems has caught up with tape [33], and disks provide better access and restore time. At the same time, the conventional wisdom that data expands to fill storage space is proving to be untrue. Douceur's examination of nearly five thousand machines finds that file systems are now only 53% full, on average [19]. Furthermore, the amount of newly written data per client per day is a small fraction of the total file system [43, 45, 50]. Several systems take advantage of low write traffic in the presence of excess storage capacity, including Elephant [43] and S4 [47].

Pastiche uses some of this excess disk capacity for efficient, effective, and administration-free backup. Pastiche nodes form a cooperative—though untrusted—collection of machines that provide mutual backup services. Because individual machines may come and go [6], each Pastiche node must replicate its archival data on more than one peer. Most of these replicas are placed nearby to ease network overhead and minimize restore time, though at least one replica must be elsewhere to guard against catastrophe. With no effort on the part of the user and modest additional disk space, backups are provided automatically. Pastiche is primarily aimed at end-user machines, but it can be used for back-end repositories with some care.

Pastiche cannot afford to keep duplicate copies of data on each replica. Luckily, much of the data on a given machine is not unique, and is generated at install time. Furthermore, for most machines, common data will be shared widely. The default installation of Office 2000 Professional requires 217 MB; it is nearly ubiquitous and different installations are largely the same. Randomly grouping disparate file systems and coalescing duplicate files produces significant savings [5]. Pastiche identifies systems with overlap to increase this savings.

Pastiche builds on three recent developments to accomplish its goals. Pastry [41], a peer-to-peer network, provides scalable, self-administered routing and node location. Content-based indexing [27, 30] provides flexible discovery of redundant data within similar files. Convergent encryption [6] allows hosts to use the same encrypted representation for common data without sharing keys.

Even with these building blocks, Pastiche still faces a number of challenges. How can nodes discover *backup buddies* with substantial overlap without a centralized directory? How can nodes reuse their own on-disk state to backup others? How can nodes restore files—or an entire machine—without requiring administrative intervention? How can nodes detect unfaithful buddies?

Pastiche computes a small *abstract* of a file system's content that potential backup buddies can inspect to approximate overlap. Pastiche is able to limit the size of the abstract by taking advantage of the fact that arbitrary, small pieces of larger logical entities are almost always unique and can, therefore, stand for the whole. This allows machines with common installations to find suitable buddies with very little effort. Machines with uncommon installations may need to use a Pastry overlay with a new routing metric, *coverage rate.*

Because sharing is supported at a sub-file granularity, Pastiche provides a new file system, *chunkstore*. Chunkstore stores all data—the host's as well backup state—in the units of sharing, without compromising the performance of common-case workloads.

Archive state is described by a *skeleton* tree of metadata. The root of this tree can be recovered from the Pastry overlay with only the name and passphrase of the machine to be restored. Entire file systems are restored as easily as a single file.

To address the problem of storing data on untrusted nodes, Pastiche uses a probabilistic mechanism to detect missing backup state by periodically querying buddies for stored data. Pastiche is able to keep the overhead of these queries small, bounding the chance of loss.

An examination of file system data shows that abstracts of a few hundred bytes effectively discriminate between candidate buddies. Simulations show that Pastiche nodes with common installations can easily find others with good overlap. The chunkstore file system induces overhead of 7.4% on a modified Andrew Benchmark, despite its unoptimized layout. Finally, analytical results show that a Pastiche node can detect corrupted backup state with high probability by checking about 0.1% of all chunks.

2 Enabling Technologies

Pastiche depends on three enabling technologies. The first is Pastry, a scalable, self-organizing, peer-to-peer routing and object location infrastructure [41]. The second is content-based indexing [27, 30], a technique that finds common data across different files. The third is convergent encryption [6], which allows sharing without compromising privacy. The remainder of this section describes each of these, focusing on the features essential to Pastiche.

2.1 Peer-to-Peer Routing

Pastiche eschews the use of a centralized authority to manage backup sites. Such an authority would be a single point of control, limiting scalability and increasing expense. Instead, Pastiche relies on Pastry, a scalable, self-organizing, routing and object location infrastructure for peer-to-peer applications.

Each Pastry node is named by a *nodeId*; the set of all nodeId's are expected to be uniformly distributed in the nodeId space. Any two Pastry nodes have some way of measuring their *proximity* to one another. Typically, this metric captures some notion of network costs.

Each node N maintains three sets of state: a *leaf set*, a *neighborhood set*, and a *routing table*. The leaf set consists of L nodes; $L/2$ are those with the closest numerically smaller nodeIds, and $L/2$ are the closest larger ones. The neighborhood set of M nodes contains those closest to N according to the proximity metric. The Pastry group has deprecated the neighborhood set. However, as we show in Section 5.3, the neighborhood set is critical to buddy discovery for nodes with uncommon installations.

The routing table supports *prefix routing*. There is one row per hexadecimal digit in the nodeId space. The first row contains a list of nodes whose nodeIds differ from the current node's in the first digit; there is one entry for each possible digit value. The second row holds a list of nodes whose first digit is the same as the current node's, but whose second digit differs. To route to an arbitrary destination, a packet is forwarded to the node with a matching prefix that is at least one digit longer than that of the current node. If such a node is not known, the packet is forwarded to a node with an identical prefix, but that is numerically closer to the destination in nodeId space. This process continues until the destination node appears in the leaf set, after which it is delivered directly. The expected number of routing steps is $\log N$, where N is the number of nodes.

Many positions in the routing table can be satisfied by more than one node. When given a choice, Pastry records the closest node according to the proximity metric. As a result, the nodes in a routing table sharing a shorter prefix will tend to be nearby since there are many such nodes. However, any particular node is likely to be far.

Pastry is self-organizing; nodes can come and go at will. To maintain Pastry's locality properties, a new node must join with one that is nearby according to the proximity metric. Pastry provides a *seed discovery protocol* that finds such a node given an arbitrary starting point [10]. Pastiche uses two separate Pastry overlay networks, but uses them only during buddy discovery.

Once a node has identified its backup set, all further traffic is routed directly via IP.

Pastiche adds two mechanisms to Pastry. The first is a technique called the *lighthouse sweep* that guarantees that distinct Pastry nodes are queried during buddy discovery. The second is a distance metric based on file system contents; this is used to find buddies for machines with rare installations.

2.2 Content-Based Indexing

To minimize storage overhead, Pastiche must find redundant data across versions of files, files in a system, and files on distinct machines. Rsync [48] and Tivoli [1] employ schemes for finding common subsets in two versions of (presumably) the same file. However, these techniques cannot easily capture general sharing.

The challenge is to find sharing—and hence structure— across seemingly unrelated files *without* knowing the underlying structure. Content-based indexing accomplishes this by identifying boundary regions, called *anchors* [27], using Rabin fingerprints [39]. A fingerprint is computed for each overlapping k-byte substring in a file. If the low-order bits of a fingerprint match a predetermined value, that offset is marked as an anchor. Anchors divide files into *chunks*. Since anchors are purely content-driven, editing operations change only the chunks they touch, even if they change offsets.

As with LBFS [30], Pastiche names each chunk by taking a SHA-1 hash [32] of its contents. The probability that two different chunks will hash to the same value is much lower than the probability of hardware errors. It is therefore customary to assume that chunks that hash to the same value are in fact the same chunk, and Pastiche adopts this custom. In Pastiche, these chunks form the basis of on-disk file structures in chunkstore to easily share data between local host and remote client.

2.3 Sharing with Confidentiality

A well-chosen backup buddy has much of a Pastiche node's data, even before the first backup. However, Pastiche must guarantee the confidentiality and integrity of its participants' data. If clients are free to choose their own cryptographic keys, chunks with identical content will be represented differently, precluding sharing.

The Farsite file system solves this problem with convergent encryption [6]. Under convergent encryption, each file is encrypted by a key derived from the file's contents. Farsite then encrypts the file key with a *key-encrypting key*, known only to the client; this key is stored with the file. As a file is shared by more clients, it gains new encrypted keys; each client shares the single encrypted file.

Pastiche applies convergent encryption to all on-disk chunks. If a Pastiche node backs up a new chunk not already stored on a backup buddy, the buddy cannot discover its contents after shipment. However, if the buddy has that chunk, it knows that the node also stores that data. Pastiche allows this small information leak in exchange for increased storage efficiency.

3 Design

Pastiche data is stored on disk as chunks. Chunk boundaries are determined by content-based indexing, and encrypted with convergent encryption. Chunks carry *owner lists*, which name the set of nodes with an interest in a chunk. Chunks may be stored on a machine's disk for that machine, a backup client, or both. Data chunks are immutable, and each chunk persists until no node holds a reference to it. Pastiche ensures that only rightful owners are capable of removing a reference to (and possibly deleting) a chunk.

When a newly written file is closed, it is scheduled for chunking. Each chunk c is hashed; the result is called the chunk's *handle*, H_c. Each handle is used to generate a symmetric encryption key, K_c, for its chunk. The handle is hashed again to determine the public chunkId, I_c, of the chunk. Each chunk is stored on disk encrypted by K_c and named by I_c. This process is illustrated in Figure 1.

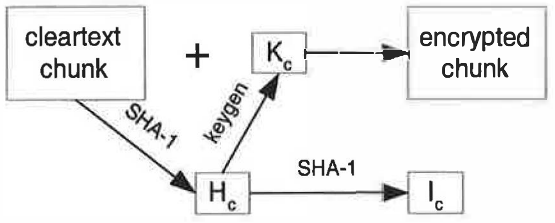
Before writing a chunk to disk, Pastiche first checks to see if it already exists. If so, the local host is added to the owner list if necessary, and the local reference count is incremented. Otherwise, the chunk is encrypted, a message authentication code [31] is appended, and the chunk is written out to disk with a reference count of one for the local owner.

Chunking and writing to disk are deferred to avoid needless overhead for files with short lifetimes [50], at the cost of slightly weaker persistence guarantees. The list of chunkIds that describes a node's current file system is called its *signature*.

Data chunks are immutable. When a file is overwritten, its set of constituent chunks may change. Any chunks no longer part of the file have their local owner's reference count decremented; if the reference count drops to zero, the local owner is removed. If the owner list becomes empty, the chunk's storage is reclaimed. File deletion is handled similarly.

The meta-data for a file contains the list of handles for the chunks comprising that file, plus the usual contents: ownership, permissions, creation and modification times, etc. The handles in this list are used to derive the decryption key and chunkId for each constituent chunk.

Meta-data chunks are encrypted to protect the handle values and hence cryptographic keys. This differs slightly from Farsite's use of convergent encryption.



This figure depicts how chunks are stored and named. A cleartext chunk is hashed, producing its handle. The handle is used for key generation, and hashed again to produce the chunkId. The chunk is stored encrypted by the key and named by the chunkId.

Figure 1: Naming and Storing Chunks

Farsite stores keys with data, encrypting each derived key with a key private to the writing host. Pastiche stores handles, and hence keys, in the meta-data blocks.

Unlike data, meta-data is is not chunked and is mutable. Pastiche does not chunk meta-data because it is typically small and unlikely to be shared. Meta-data is mutable to avoid cascading writes. Each write to a file changes its constituent chunkIds. If meta-data were immutable, Pastiche would have to create a new meta-data chunk with a new name for every update. This new name would have to be added to the enclosing directory, which would also change, and so on to the file system root. Instead, the H_c, K_c, and I_c for a file's meta-data are computed only at creation time, and are re-used thereafter.

The meta-data object corresponding to a file system root is treated specially: its H_c is generated by a host-specific passphrase. As Section 3.4 explains, this passphrase plus the machine's name is all that is required to restore a machine from scratch.

A chunk that is part of another node's backup state includes that nodeId in its owner list. Remote hosts supply a public key with their backup storage requests. Requests to remove references must be signed by the corresponding secret key, otherwise those requests are rejected. This prevents third-party deletions, though it does not prevent the buddy from dropping chunks of its own accord.

Storing files directly as chunks simplifies a number of Pastiche's tasks and imposes modest performance costs. It simplifies the implementation of chunk sharing, convergent encryption, and backup/restore. Without chunkstore, Pastiche would have to keep a persistent index consistent with on-disk files. This index would have to be consulted during backup and restore, and complicates garbage collection of chunks retired during snapshot. Furthermore, convergent encryption requires that each chunk be encrypted separately, complicating a contiguous layout. The only alternative would be to detect sharing only at the file level, with a corresponding increase in storage costs for backup.

3.1 Abstracts: Finding Redundancy

Much of the long-lived data on a machine is written once and then never overwritten. Observations of file type [19] and volume ownership [45] suggest that the amount of data written thereafter will be small. In other words, the signature of a node is not likely to change much over time. Therefore, if all data had to be shipped to a backup site, the initial backup of a freshly installed machine is likely be the most expensive.

An ideal backup buddy for a newly-installed Pastiche node is one that holds a superset of the new machine's data; machines with more complete coverage are preferred to those with less. One simple way to find such nodes is to ship the full signature of the new node to candidate buddies, and have them report degree of overlap.

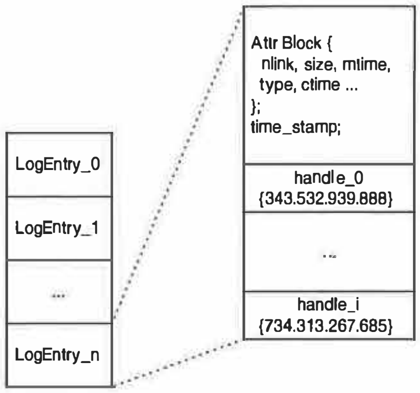
Unfortunately, signatures are large: 20 bytes per chunk. Expected chunk size is a function of how anchors are selected. In our implementation, this size is 16 KB, so signatures are expected to cost about 1.3 MB per GB of stored data. If this cost were paid only once, it might be acceptable. However, a node's buddy set can change over time as buddies are found to be unreliable or as degrees of overlap change.

Rather than send a full signature, Pastiche nodes send a small, random subset of their signatures called an *abstract*. This is motivated by the following observation: most data on disk belongs to files that are part of a much larger logical entity. For example, a Linux hacker with the kernel source tree has largely the same source tree as others working on the same version. Any machine holding even a small number of random chunks in common with this source tree is likely to hold most of them. Preliminary experiments show that tens of chunkIds—a few hundred bytes—are enough to distinguish good matches from bad ones. This size is similar to that reported by Border for individual web objects [8].

3.2 Overlays: Finding a Set of Buddies

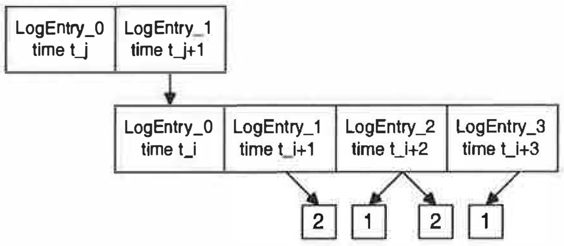
All of a node's buddies should have substantial overlap with it to reduce storage overhead. In addition, most buddies should be nearby to reduce global network load and improve restore performance. However, at least one buddy must be located elsewhere to provide geographic diversity. As a rule of thumb, each Pastiche node maintains five buddies.

Pastiche uses two Pastry overlays to facilitate buddy discovery. One is a standard Pastry overlay organized by network proximity. The other is organized by file system overlap. Every Pastiche node joins a Pastry overlay organized by network distance. Its nodeId is a hash of the machine's fully-qualified domain name. Once it has joined, the new node picks a random nodeId and routes



This figure depicts how a meta-data chunk is stored on disk. The chunk is stored as a log of file states, where each entry in the log represents the state of the file after the update. Entries are comprised of an attribute block, a time stamp, and a list of constituent chunk handles.

Figure 2: Meta-data Chunk Layout



This figure depicts a small skeleton. Each chunk is stored as a log, and each entry in the log has references to other chunks. The top chunk begins empty, and then adds another. The bottom chunk adds a data chunk, appends another to the end, and then removes the first chunk.

Figure 3: Chunk Skeleton

a discovery request to it. The discovery request contains the new node's abstract. Each node encountered on the route computes its *coverage*–the fraction of chunks in the abstract stored locally–and returns it.

If the initial probe does not generate a sufficient candidate set, the probe process is repeated. Subsequent probes are generated by varying the first digit of the original nodeId. Since Pastry uses prefix routing, each probe will generate sets of candidates disjoint from those already examined. We call this rotating probe a *lighthouse sweep*.

Nodes with common installations should find a sufficient candidate set easily. However, nodes with rare installations will have more difficulty. Nodes that do not find an adequate set during a lighthouse sweep join a second overlay, called the *coverage-rate* overlay. This overlay uses file system overlap rather than network hops as the distance metric. The new node chooses backup buddies from its Pastry neighbor set—the set of nodes encountered during join with the best coverage available.

The use of coverage rate as a distance metric has interesting implications for Pastry. Like network distance, coverage rate does not obey the triangle inequality. Unlike network distance, coverage rate is not symmetric; if A holds all of B's files, the converse is probably not true. This means that an individual node must build its routing state based on the correct perspective. Likewise, the seeding algorithm must be supplied with the new node's abstract, so that it can compute coverage from the correct point of view.

It is possible for a malicious node to habitually under- or over-report its coverage. If it under-reports, it can avoid being selected as a buddy. If it over-reports, it can attract unsuspecting clients only to discard their backup state. Unfortunately, this is possible no matter who computes coverage rates. An honest node can be given a random list of chunkIds as an abstract; such an abstract is unlikely to match anything. Likewise, a malicious node can cache and report abstracts sent by others with commonly-appearing chunkIds, hoping for a false match.

3.3 Backup Protocol

A Pastiche node has full control over what, when, and how often to back up. Each discrete backup event is viewed as a single snapshot. Nodes can subscribe to a calendar-based cycle, a landmark-based scheme [43], or any other schedule. Because a machine is responsible for its own archival plan, it keeps a meta-data *skeleton* for each retained snapshot. A file that is not part of the local machine's current file system, but is part of one or more archived snapshots, has a corresponding meta-data entry stored on the local machine.

The skeleton for all retained snapshots is stored as a collection of persistent, per-file logs, as shown in Figures 2 and 3. The skeleton representing a machine's current file system state plus all retained snapshots is stored both on the machine and all of its backup buddies.

The state necessary to establish a new snapshot consists of three things: the chunks to be added to the backup store, the list of chunks to be removed, and the meta-data objects in the skeleton that change as a result. We call these the add set, delete set, and meta-data list.

The snapshot process begins by shipping the host's public key. This key will be associated with any new chunks to validate later requests to delete or replace them. The snapshot node then forwards the chunkIds for elements of the add set. If any of those chunks are not already stored on the buddy, the buddy fetches the chunks from the node.

Next, the node sends the delete list. The snapshot host adds a chunkId to the delete list only if it does not belong to any snapshot the client wishes to retain. The delete list must be signed, and this signature is checked to ensure it matches the public key associated with any chunks scheduled for deletion. Note that deletion is not effected immediately. It is deferred to the end of the snapshot process.

Finally, the snapshot node sends any updated meta-data chunks. Since they may overwrite old meta-data chunks, their chunkIds must also be signed. When all state has been transferred, the host requests a *commit* of the checkpoint. Before responding, the buddy must ensure that all new chunks, changed meta-data objects, and deleted chunkIds are stored persistently. Once that is complete, the buddy can respond, and later apply the new snapshot by performing the appropriate deletions.

The performance of snapshots is not crucial, since they are asynchronous. The only exception is marking chunkstore copy-on-write, which must be done synchronously. However, as with AFS's volume clone operation [23], this is inexpensive. The load induced on a buddy by the backup protocol can be regulated with resource containers [2] or progress-based mechanisms [20]. This load is quantified in Section 5.2.

The snapshot process is restartable. The most expensive phase—shipping new data chunks—makes progress even in the presence of failures, since new chunks are stored as they arrive. After the new snapshot is applied, a faithful buddy will have a complete copy of the meta-data skeleton, plus all data chunks the skeleton names.

3.4 Restoration

A Pastiche node retains its archive skeleton, so performing partial restores is straightforward. The node identifies which chunks correspond to the restore request, and obtains them from the nearest buddy.

Recovering the entire machine requires a way to bootstrap the skeleton. To do so, a Pastiche node keeps a copy of its root meta-data object on each member of its network-distance leaf set. When a machine must recover from disaster, it rejoins the distance-based overlay with the same nodeId, which is computed from its host name. It then obtains its root node from one of its leaves, and decrypts it with the key generated from the host's passphrase. Since the root block contains the set of buddies in effect when it was replicated, the node can recover all other state.

3.5 Detecting Failure and Malice

A buddy is expected to retain backup snapshots, but is not required to do so. When faced with a sudden disk space crisis, a buddy is free to reclaim space. A buddy may also fail or be connected intermittently, leaving its ability to serve future restore requests in doubt. Finally, a malicious buddy may claim to store chunks without actually doing so.

Pastiche employs a probabilistic mechanism to detect all of these situations. Before taking a new snapshot, each Pastiche node asks buddies for a random subset of chunks from the node's archive. By requesting a modest number of chunks, clients can bound the probability that compromised backup state goes undetected. Savvy users of traditional backup schemes already employ this technique as an end-to-end confirmation of correctness. If a buddy cannot produce data it claims to hold, the client removes it from its buddy list and initiates a search for a replacement. If a buddy has not responded for a significant period of time, the client likewise removes it.

Unfortunately, this request provides only instantaneous assurance; a malicious node can drop chunks after they are requested. Thus, increasing the frequency of requests does not provide increased assurance, while increasing the size of a single request does.

This technique assumes that a malicious party cannot occupy a substantial fraction of the nodeId space, and hence cannot produce collusion between a single host's backup buddies. Defending against such *Sybil attacks* [18] in practice requires some centralized agency to certify identities [9]. Such certification is essential in Pastiche, because nodes can increase their chances of being selected as buddies by falsely over-reporting their coverage rates.

Pastiche leverages the spot-check mechanism to detect snapshots belonging to decommissioned machines. Each buddy knows that its corresponding host does not fully trust it. So, the buddy expects to be probed periodically. If a buddy does not hear from its corresponding host for an exceptionally long period, it can assume that the host has either been decommissioned or re-installed. This decision must be made very conservatively, lest a long-lived failure be mistaken for a voluntary removal.

This lack of trust places some limits on Pastiche's applicability; it is intended primarily for end-user systems for which backup is currently difficult. Back-end repositories are already centrally managed, and adding backup services to them is comparatively simple. However, Pastiche can be extended to such services by choosing one or more backup buddies to be an administered, trusted machine, provided the service's expected workload shows temporal locality. Without such locality, the performance of chunkstore is likely to be poor.

3.6 Preventing Greed

A greedy host can aggressively consume space by using storage on many hosts and never retiring any of them. This is the most challenging problem faced by Pastiche. Pastiche needs a distributed quota enforcement mechanism; a node should occupy only as much space as it contributes. We have considered three solutions to this problem, but none are completely satisfactory.

The first solution places nodes into equivalence classes based on the resources they consume. Each node monitors the overall storage costs imposed by its backup clients, and compares these costs to its own usage. Those that are much more space-intensive are ejected, and must search for a more suitable partner. Unfortunately, this mechanism is circumvented by the Sybil attack.

The second approach is to force each node to solve cryptographic puzzles [24] in proportion to the amount of storage it occupies. Forging identities is no defense against this, nor is spreading snapshots across more than the usual number of buddies. However, we dislike this solution for several reasons. First, it adds needless expense to backup, which is antithetical to Pastiche's goals. Second, it trades something other than storage for storage space. Third, not all nodes will have equivalent processing power, so it is difficult to provision the solution properly.

The third approach is to account for space with some form of electronic currency [12]. It is sufficient to use an offline protocol [4]; some amount of double-spending is tolerable as long as abusers can be detected eventually. However, currency accounting requires that backup be *goods atomic* [49]; the exchange of currency and backup state must be an atomic transaction. Adding this complicates Pastiche substantially.

3.7 An Alternative Design

Before settling on Pastiche's current approach, we considered an alternative that appears to be a more natural fit to a peer-to-peer substrate. Instead of having a small list of backup buddies, each holding a complete backup, this alternative stores each chunk on the K Pastry nodes with nodeIds numerically closest to the chunk's identifier. We call this alternative the *fine-grained approach.*

The fine-grained approach has two advantages over Pastiche. First, it ensures that only K backup copies of a chunk exist anywhere in the network. Second, Pastry takes care of detecting failed or unresponsive hosts, and individual nodes need not keep track of them.

However, the fine-grained approach also has two disadvantages. The first is the loss of network proximity for most replicas, increasing network load during backup and latency during restoration. Restoration costs can be avoided by caching along the Pastry route taken by backup chunks, but this increases global disk overhead.

The second disadvantage is the difficulty in dealing with malicious nodes. It is much harder for a client to probe for malicious nodes, since the set of nodes containing client state is on the order of the number of chunks. Pastiche trades disk space to reduce network costs and give clients the tools to ensure that their backups are safely stored.

4 Implementation

The Pastiche prototype consists of two main components: the chunkstore file system and a backup daemon. Chunkstore is written in C and is implemented primarily in user space for simplicity. The user-space component is called `pclientd`. A small, in-kernel portion implements the vnode interface [26], integrating chunkstore with Linux 2.4.18. Pastiche uses the XFS device from Arla [51], an open source AFS implementation, for this in-kernel portion.

Data is stored as individual chunks in an underlying file system. For performance reasons, Pastiche also maintains a cache of contiguous, decrypted copies of recently used files, called *container files*. Our prototype does not yet support whole-machine backup, because we have not implemented booting a kernel from chunkstore.

The XFS device sees only container files, and `pclientd` acts as mediator between the device, the container files, and chunkstore. When an application requests a file that is not in a container, `pclientd` retrieves the meta-data chunk for that file from chunkstore and uses it to form a contiguous container file. `pclientd` then returns the inode of the container file to the device, and subsequent operations are applied to the container. The container file cache is managed with LRU replacement, given a maximum size.

`pclientd` is notified of each `close`. If the corresponding file is dirty, it is scheduled for chunking. Chunking is deferred for 30 seconds, to avoid needless overhead for short-lived files [50]. We implemented convergent encryption using the `openssl-0.9.7-beta3` cryptographic library. Each chunk was encrypted using a 128-bit key and the AES stream cipher [17].

Container files restore the parity between logical and on-disk proximity that storing chunks individually eliminates. However, storing chunks individually still induces some storage overhead. By storing each chunk separately, Pastiche files will yield more internal fragmentation than if they had been stored contiguously. Recall that content-based indexing generates chunks by examining the lower k bits in a Rabin fingerprint; if these bits match some target value, that offset is marked as a chunk boundary. On average, one would expect to lose half of a disk block per chunk. So, we set k to 14, giving an expected chunk size of 16KB and expected fragmentation overhead to 3.1%.

Meta-data chunks are stored as a log of updates to the file. Each time a file is re-chunked, the list of its constituent chunks is appended to the log. Deletion is represented with a terminal log entry. `pclientd` only appends to these logs, and thus never removes a chunk from chunkstore.

The backup daemon, called `backupd`, is written in C and uses the rpc2 remote procedure call package for communication [44]. It acts as both the backup server and client. The server manages remote requests for storage and restoration, while the client supervises selection of buddies and snapshots. Additionally, `backupd` cleans meta-data logs and reaps deleted chunks.

`backupd` communicates with `pclientd` through file locking of on-disk chunks. This is simple and can be efficient, since `backupd` need not hold all locks to guarantee a consistent snapshot. Once the root meta-data chunk is read, all reachable chunks are guaranteed to remain reachable, since none of them will be deleted by `pclientd`. Some meta-data chunks may still become tainted [25] with additional log entries. However, these entries can be detected via timestamps during backup and restore, rendering their inclusion in a snapshot harmless.

We also provide several utilities to allow users to manage the file system: `forcesnap` forces `backupd` to take a system snapshot immediately, `forcechunk` forces `pclientd` to chunk all files in its queue immediately, and `rfile` restores a file or subtree to a previous state. Each utility communicates with `pclientd` and `backupd` through Unix domain sockets.

5 Evaluation

In evaluating our prototype, we set out to answer the following questions:

- What is the performance of the file system? Which operations perform well and which perform badly?
- How long do backups and restores take?
- How large must fingerprints be? Is the lighthouse sweep able to find buddies?
- Does the coverage-rate overlay yield suitable backup buddies?
- Are the costs to detect malicious nodes reasonable?

All experiments were run on machines with a 550 MHz Pentium III Xeon processor, 256MB of memory, and a 10k RPM SCSI Ultra wide disk, with 4.7 ms seek time, 3.0 ms rotational latency, and 41 MB/s peak throughput.

AB phase	ext2fs		chunkstore	
`mkdir`	1.23	(0.04)	1.03	(0.05)
`cp`	3.47	(0.28)	6.26	(0.16)
`scandir`	0.0	(0)	0.03	(0)
`cat`	1.75	(0.02)	2.23	(0.02)
`make`	38.62	(0.45)	38.88	(0.5)
total	45.08	(0.39)	48.43	(0.58)

This figure presents the results of a modified Andrew Benchmark. Times are reported in seconds, and standard deviations are given in parentheses.

Figure 4: Andrew Benchmark

Task	ext2fs		chunkstore	
`wide create`	2.44	(0.06)	6.99	(0.02)
`wide mkdir`	2.30	(0.02)	6.31	(0.03)
`deep mkdir`	4.07	(0.02)	5.64	(0.01)
`bulk xfer`	12.79	(0.01)	12.75	(0.02)

This figure presents the results of file creation and I/O throughput benchmarks. Times are reported in seconds, and standard deviations are given in parentheses.

Figure 5: Micro-benchmarks

5.1 Performance

What is the overhead induced by chunkstore? To answer this, we compare the performance of chunkstore to the underlying, native file system, `ext2fs`. We measure this overhead with a modified Andrew Benchmark [23]. Our benchmark is identical to the original in form, but uses the `apache 1.3.26` source tree. This source tree is 9.6MB in size; when compiled, the tree occupies 12MB.

We ran five trials; the results are shown in Figure 4. While the `make` step is not I/O bound, it does experience slight overhead. This is due in part to the cost of computing the Rabin fingerprints of the copied tree and the extra cost of creating and deleting files. The copied data is scheduled to be chunked when written, and 30 seconds later—during the `make` step—chunking begins.

The total overhead of 7.4% is reasonable, though the `copy` phase is expensive; it takes 80% longer in chunkstore. We believe that this overhead is due to excess meta-data management in chunkstore, rather than limits on peak I/O throughput. To confirm our hypothesis, we performed several micro-benchmarks to isolate the operations involved in copying a source tree - writing data and creating files.

To examine chunkstore's performance when creating files and directories, we ran three different experiments—`wide create`, `wide mkdir`, and `deep mkdir`. In `wide create`, 1000 new files were created in the same directory. In `wide mkdir` 1000 new directories were created in the same directory,

Task	time	
cp	6.26	(0.07)
backup	6.55	(0.01)
rm	1.24	(0.01)
restore	5.54	(0.07)
nfs cp	3.76	(0.16)

This figure presents the results of the backup and restore experiment. Times are reported in seconds, and standard deviations are given in parentheses.

Figure 6: Backup and Restore

and in deep mkdir 1000 new directories were made recursively inside of one another. We again ran five trials of each; the results are in Figure 5.

wide create and wide mkdir each ran about 186% and 174% slower than ext2fs, respectively, while deep mkdir ran about 38% slower. Chunkstore's poor performance is due to meta-data chunk and container file maintenance. When chunkstore creates a file, it must update or create three files: a new meta-data chunk, a new container file, and the parent meta-data chunk.

The deep mkdir experiment shows that the number of entries in the parent directory is also significant. This is because of the way directory entries are laid out in the meta-data chunks and the container files. In both cases, directory entries are stored in a linear array. Our current implementation rewrites the entire list to the container file and chunk whenever a new entry is added. During deep mkdir, there is only ever one entry in the list, which makes creating a file faster.

It is also interesting to note that wide mkdir is somewhat faster than wide create. The reason for this is related to how the in-kernel XFS device handles file and directory creation. When a regular file is created, the XFS device makes an extra upcall to pclientd to close the newly created file, and does not make this call when a directory is created.

To further verify that I/O throughput was not responsible for chunkstore's slow copy phase in the modified Andrew Benchmark, we also ran a bulk xfer experiment. In this experiment, a new file was created, 256MB of data were written to it, and then the file was closed. As before, we ran five trials; the results are in Figure 5. Chunkstore and ext2fs performed within 1% of each other, meaning that their I/O throughput are statistically identical.

5.2 Backing Up and Recovering a File System

To determine the performance of our backup and restore utilities, we applied them to a file system consisting of the openssl-0.9.7-beta3 source tree. This 13.4MB tree of 1641 files and 109 directories is stored in Pastiche as 4004 chunks.

Each of five trials consisted of four phases - copying the source tree into the file system, sending it to a backup buddy, removing the local source tree, and restoring the source tree from the backup buddy. Pastiche's backup and restore performs comparably to the time to copy the source tree over NFS. The results are in Figure 6.

It should also be noted that the demand on resources the buddy experiences while carrying out backup and restore is very bursty. During the five trials, backupd used a maximum of 8MB of memory, averaged 12 disk transfers/sec with a maximum of 414 transfers/sec, and averaged a 70% idle CPU with a minimum of 13%.

5.3 Finding Buddies

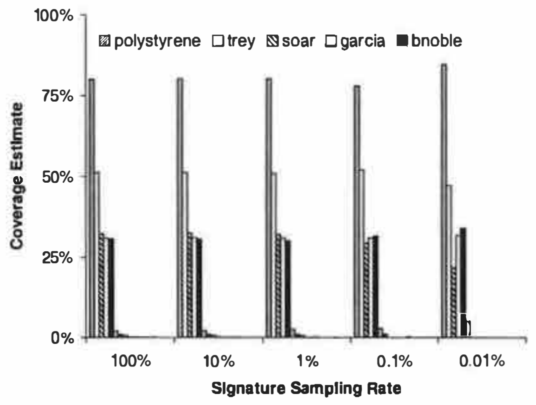
Next we turn our attention to the buddy discovery process. There are three questions to answer. First, how large must an abstract be to discriminate good buddy candidates from bad ones? Second, how effective is the lighthouse sweep in discovering buddies? Third, how effective is the coverage-rate overlay in discovering buddies? To answer the first question, we took the signatures of seventeen machines at Michigan. These machines run Windows, Linux, Solaris, and various flavors of BSD. We also took the signatures of two freshly installed machines.

The first ran Windows 98 with an Office 2000 Professional installation, but without any service packs applied. This machine held roughly 90 thousand chunks[1]. The second was a Linux machine running a Debian unstable release, configured as a conventional workstation with development and document processing tools. This machine held approximately 270 thousand chunks. We chose this machine as a worst case. Some of our comparison machines are Debian, but only one runs the unstable distribution. This distribution changes quickly, and this machine is updated infrequently.

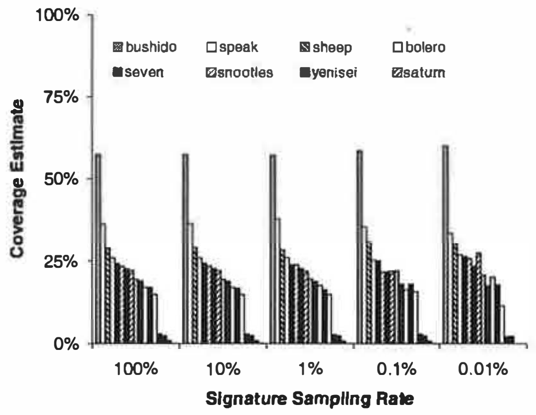
We computed the actual coverage for each of these machines given full signatures. To estimate the impact of smaller abstracts on coverage estimates, we took uniform random samples of the signature at rates of 10%, 1%, 0.1%, and 0.01%; there are six samples at each rate.

The results for the Windows 98 machine are plotted in Figure 7(a). The x axis gives sampling rate, and the y axis shows coverage rate. The 100% "sample" shows exact coverage; each of the others is an estimate given a smaller sample. Each group of bars represents the coverage estimate for each of the seventeen hosts. Within each group, the hosts are sorted by actual coverage rate,

[1]For this experiment, we used a smaller expected chunk size of 4KB; Pastiche's larger chunks may require slightly larger sampling rates.



(a) Windows 98/Office 2000



(b) Debian Developer

Figure 7: Varying Abstract Size

from highest to lowest. The top five matches are identified in the legend. `polystyrene` is a Win98 machine running Office 2000, with all relevant service packs and security updates applied.

The estimates are surprisingly independent of sample size; the lowest rate produces abstracts of around 10 chunkIds. Only `soar`'s estimate changes appreciably. However, its coverage rate is comparable to `garcia`'s and `bnoble`'s; choosing either of the latter in favor of `soar` is of no consequence.

Figure 7(b) shows the results for our Linux machine, the top eight matches are identified in the legend. The overall match rates are lower, but machines with other distributions still have substantial matches. As with the Windows 98 host, coverage estimates do not change materially as abstract sizes go down. Interestingly, `bnoble`, a Windows 2000 machine, has a coverage rate for this Debian machine of almost 15%. This is because `bnoble` also has a `stable` release of Debian, installed in a VMware virtual machine; the VMware disk image is stored as a regular file in Windows. Ordinarily, files

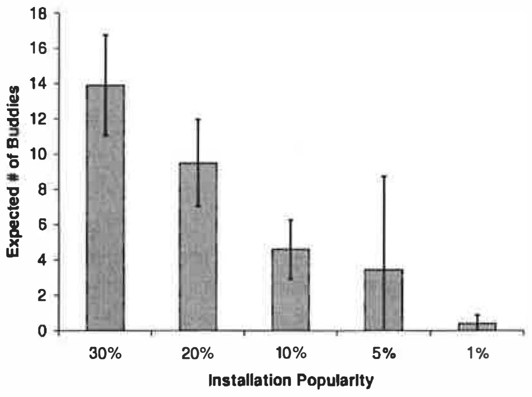


Figure 8: Expected Number of Buddies

form implicit chunk boundaries in content-based indexing. When viewed from the windows host, all of these file boundaries disappear. Despite this, content-based indexing is still able to find substantial overlap.

Small abstracts are effective only if they are delivered to a host that can provide good coverage. We conducted a simulation to determine how effectively lighthouse sweeps find useful buddies. This simulation uses SimPastry [29], a Pastry simulation/visualization tool.

For the simulation, we populated a graph with 50 thousand Pastiche nodes drawn from a distribution of 11 *types*. 30% of all nodes are the first type, types two and three each comprise 20% of all nodes, types four and five each comprise 10%, type six comprises 5%, and types seven through eleven each represent 1% of the population.

We simulated 25 different Pastry networks under these conditions. For each network, we randomly selected 1000 nodes of each type, and performed a lighthouse sweep from that node, counting the number of hosts of identical type found during the sweep. The results are shown in Figure 8. Each bar gives the average number of matches found per sweep for each category of popularity; the error bars show one standard deviation.

As expected, common nodes with representation of 10% or higher should find an adequate number of buddies on the distance overlay. Those with lower popularity will need to join the coverage-rate overlay as well. We built a Pastry simulator to determine the effectiveness of this network. Our experiments involved 10,000 nodes. Each node was assigned to one of a thousand species, one of a hundred genera, and one of ten orders. Nodes of the same order share 20% of their content; nodes of the same genus, 30%; and nodes of the same species, 70%. Only nodes of the same species can serve as backup buddies for one another.

The results of our simulations are in Figure 9. The x axis give the size of the neighborhood set, and the y

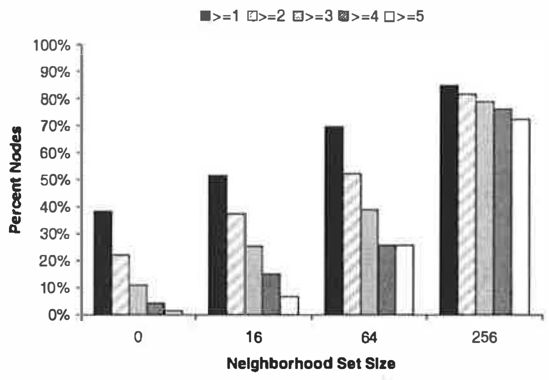


Figure 9: Coverage-rate Simulation Results

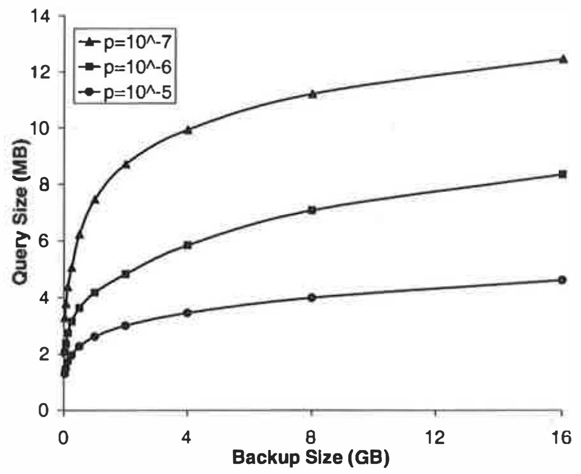


Figure 10: Growth of Query Size

axis shows the percent of all nodes who found a given number of buddies. We ran four series of trials, varying the size of the neighborhood set. We found that for a neighborhood set size of 256, 85% were able to find at least one buddy in its routing table, and 72% were able to find at least 5.

The results show that most nodes should be able to find buddies in the coverage-rate overlay. It also shows how important a role the neighborhood set plays in locating buddies. Increasing the neighborhood set from 0 to 256 increases the percent of nodes who can find at least one buddy from 38% to 85%; the percent of nodes who can find at least 5 increases from 1% to 72%.

5.4 Determining Query Size

Backup buddies can drop chunks, either in error or maliciously. If the same chunk is dropped at all replicas, the backup state is said to be *corrupted*. A node can be certain that its state is not corrupted by requesting all c chunks, but this is clearly too expensive. Instead, Pastiche nodes query just enough chunks, q, to be assured with some probability p that corrupted state will be detected if it exists.

We assume that replicas cannot collude to agree on a specific chunk to drop. Instead, each of n replicas drops chunks at some rate r. A Pastiche node must set q so that the probability of drops causing corruption and going undetected is less than or equal to p.

If r is zero, then the chance of corruption is also zero, and the problem is solved trivially. On the other hand, if r is one, then a query of one chunk is guaranteed to detect corruption. However, some intermediate values of r require queries of more chunks.

The analysis proceeds in two parts. First, we compute p_c, the probability of corruption. Second, we compute p_u, the probability that dropped chunks go undetected. These are conditionally independent for a given r, so their product expresses the event whose probability we wish to bound by p.

If each replica drops each chunk with probability r, then the chance that a chunk is dropped at all replicas is r^n, and the chance that it exists on at least one is $1 - r^n$. There are c chunks, so the chance that all of them exist on at least one replica is $(1-r^n)^c$. Therefore, the chance that at least one of them does *not* exist on *all* replicas—the chance of corruption—is:

$$p_c = 1 - (1 - r^n)^c. \tag{1}$$

Suppose a node asks a buddy for a single chunk. The chance that the buddy can supply it is $1 - r$. If the node asks for q chunks simultaneously, then the chance that the buddy can respond successfully—the chance that dropped chunks go undetected—is

$$p_u = (1 - r)^q. \tag{2}$$

We want to bound the product of these to minimize the probability of undiscovered corruption:

$$[1 - (1 - r^n)^c](1 - r)^q \leq p. \tag{3}$$

Solving for q, we get

$$q \geq \log_{1-r}\left(\frac{p}{1-(1-r^n)^c}\right). \tag{4}$$

Since r must take one of $c+1$ discrete values, it is feasible to compute the maximum q over all possible values of r, given n, c, and p.

The resulting queries grow very slowly with respect to backup size, as shown in Figure 10. This figure shows total backup size on the x axis and computed query size on the y axis, assuming an average chunk size of 16KB and five replicas. The query sizes are computed for five buddies, with three values of p: 10^{-5}, 10^{-6}, and 10^{-7}. Even for high degrees of assurance, the query costs are modest.

6 Related Work

Backup is critical, yet there is a surprisingly small amount of literature on the topic. Most work focuses on centralized backup of large installations [28, 36]; Chervenak provides a survey of a number of different backup systems [13]. Current commercial systems, such as Veritas' NetBackup, IBM's Tivoli, and Connected's remote backup service, also focus on large, centrally-managed repositories.

Several projects have suggested the use of peer-to-peer routing and object storage systems as a substrate for backup, including Chord [46], Freenet [14], and Pastry [41]. File systems built on them, such as PAST [42] and CFS [16], provide protection against machine failure. However, they do not protect against human error, nor do they provide the ability to retrieve prior versions of files once replaced. OceanStore [40], does provide these benefits, but the decision of which versions to retire rests with the utility, not its clients.

The pStore cooperative backup system [3], built on top of Chord, stores individual objects on a number nodes, rather than storing the entire set objects on a number of nodes. However, it does not exploit inter-host sharing, nor does it address the problem of hosts falsely claiming to store data. Elnikety presents a cooperative backup scheme [21] that requests random blocks from partners, but assumes that partners either drop all or none of the archived state.

A number of systems exploit duplicate data across files and machines to minimize the costs of archival storage. The Single Instance Store [5] detects and coalesces duplicate files, while Venti [38] divides files into fixed-size blocks and hashes those to find duplicate data. Neither of these approaches can take advantage of small edits that move data within a file, as content-based indexing does [27, 30]. Other sophisticated techniques for detecting such changes exist [1, 48], but must be run on pairs of files that are assumed to have overlap.

Broder provides a mathematical foundation for detecting similarity and inclusion based on sketches [8], similar to Pastiche's abstracts. Sketches of a few hundred bytes are able to find similarities among single documents on the web [7]. Pastiche extends this result, to find similarities between entire disks.

Rather than exploit redundancy, one can instead turn to the use of erasure codes [35] to stripe data across several replicas. Such codes allow for low-overhead replication, and are tolerant of the failure of one or more replicas; they are employed by Myriad [11], OceanStore [40], and Elnikety [21]. Their main shortcoming, compared to our simpler scheme, is that they require the participation of more than one node for restore.

AFS [23], Plan 9 [37], and WAFL [22] expose a snapshot primitive for a variety of purposes, including backup. Typically, snapshots are used to stage data to archival media other than disk. SnapMirror [34] leverages WAFL's snapshot mechanism to provide fine-grained, remote disk mirroring with low overhead.

7 Conclusion

Backup is tedious and expensive. Embarrassingly, the authors' own workstations are not backed up. Only their user data, stored on a distributed file system, is backed up regularly.

Pastiche enables automatic backup with no administrative costs, requiring only excess disk capacity among a set of cooperating peers. Since Pastiche matches nodes who have significant data in common, this excess capacity can be modest. Peers are selected through the use of two peer-to-peer overlay networks, one organized by network distance, the other by degree of data held in common. The self-organizing nature of these overlays, combined with mechanisms to detect failed or malicious peers, obviates the need for administrative intervention.

Evaluation of our Pastiche prototype demonstrates that this service does not penalize file system performance unduly. Simulations confirm the effectiveness of node discovery, and analysis shows that detecting malicious hosts requires only modest resources. Pastiche promises to lower the barriers to backup so that all data can be protected, not just that judged worthy of the expense and burden of current schemes.

Acknowledgements

The authors wish to thank Mark Brehob for his assistance with the analysis in Section 5.4. Jason Flinn, Jim Gray, and the anonymous reviewers provided many helpful comments.

This work is supported in part by the Intel Corporation; Novell, Inc.; the National Science Foundation under grant CCR-0208740; and the Defense Advanced Projects Agency (DARPA) and Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Intel Corporation; Novell, Inc.; the National Science Foundation; the Defense Advanced Research Projects Agency (DARPA); the Air Force Research Laboratory; or the U.S. Government.

References

[1] M. Ajtai, R. Burns, R Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the Association for Computing Machinery*, to appear.

[2] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.

[3] C. Batten, K. Barr, A. Saraf, and S. Trepetin. pStore: A secure peer-to-peer backup system. Unpublished report, MIT Laboratory for Computer Science, December 2001.

[4] M. Blaze, J. Ioannidis, and A. Keromytis. Offlinemicropayments without trusted hardware. In *Proceedings of the Fifth Annual Conference on Financial Cryptography*, Cayman Islands, BWI, February 2001.

[5] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24, Seattle, WA, August 2000.

[6] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 34–43, Santa Clara, CA, June 2000.

[7] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the 6th International World-Wide Web Conference*, pages 391–401, Santa Clara, CA, April 1997.

[8] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES*, pages 21–29, Salerno, Italy, June 1997. Published in 1998.

[9] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for structured peer-to-peer overlay networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[10] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Submitted for publication.

[11] F. Chang, M. Ji, S. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. Myriad: Cost-effective disaster tolerance. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 103–116, Monterey, CA, January 2002.

[12] D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology: Proceedings of Crypto '82*, pages 199–203, August 1982.

[13] A. L. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: A survey of backup techniques. In *Proceedings of the Joint NASA and IEEE Mass Storage Conference*, March 1998.

[14] I. Clarke, S. G. Miller, T. W. Hong, O. Sandberg, and B. Wiley. Protecting fee expression online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[15] Connected Corporation. The 60% you're missing: Preventing data loss through PC management. White paper, Farmingham, MA, 2002.

[16] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, Banff, Canada, October 2001.

[17] J. Daemen and V. Rijmen. AES proposal: Rijndael. Advanced Encryption Standard Submission, 2nd version, March 1999.

[18] J. R. Douceur. The Sybil attack. In *1st International Workshop on Peer-to-Peer Systems*, Cambridge, MA, March 2002.

[19] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 59–70, Atlanta, GA, May 1999.

[20] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 247–260, Kiawah Island Resort, SC, December 1999.

[21] S. Elnikety, M. Lillibridge, M. Burrows, and W. Zwaenepoel. Cooperative backup system. In *The USENIX Conference on File and Storage Technologies*, Monterey, CA, January 2002. Work-in-progress report.

[22] D. Hitz, J. Lau, and M. A. Malcom. File system design for an NFS file server appliance. In *Proceedings USENIX Winter Technical Conference*, pages 235–246, San Francisco, CA, January 1994.

[23] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[24] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the Network and Distributed System Security Symposium*, pages 151–165, San Diego, CA, February 1999.

[25] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance

and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, Saint Malo, France, October 1997.

[26] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Association Summer Conference Proceedings*, pages 238–247, Atlanta, GA, June 1986.

[27] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Conference*, pages 1–10, San Francisco, CA, January 1994.

[28] E. Melski. Burt: the backup and recovery tool. In *Proceedings of LISA'99*, pages 207–217, Seattle, WA, November 1999.

[29] Microsoft Corporation. SimPastry. http://www.research.microsoft.com/~antr/Pastry/ download.htm.

[30] A. Muthitacharoen, B. Chen, and D. Maziéres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, Banff, Candada, October 2001.

[31] National Institute of Standards and Technology. Computer data authentication. FIPS Publication #113, May 1985.

[32] National Institute of Standards and Technology. Secure hash standard. FIPS Publication #180-1, April 1997.

[33] Network Appliance. NetApp unveils first nearstore release. *Computer Reseller News*, page 33, March 25, 2002.

[34] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: File system based asynchronous mirroring for disaster recovery. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 117–129, Monterey, CA, January 2002.

[35] W. W. Peterson and E. J. Weldon. *Error-correcting Codes*. The MIT Press, 1972.

[36] W. C. Preston. Using Gigabit Ethernet to backup six Terabytes. In *Proceedings of LISA'98*, pages 87–95, Boston, MA, December 1998.

[37] S. Quinlan. A cache WORM file system. *Software—Practice and Experience*, 21(12):1289–1299, December 1991.

[38] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 89–102, Monterey, CA, January 2002.

[39] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[40] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, September 2001.

[41] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, Heidelberg, Germany, November 2001.

[42] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, Banff, Canada, October 2001.

[43] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, Kiawah Island Resort, SC, December 1999.

[44] M. Satyanarayanan. *RPC2 User Guide and Reference Manual*. School of Computer Science, Carnegie Mellon University, October 1991.

[45] M. Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, 14(2):200–222, May 1996.

[46] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference*, pages 149–160, San Diego, CA, August 2001.

[47] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 165–179, San Diego, CA, October 2000.

[48] A. Tridgell. *Efficient algorithms for sorting and synchronization*. PhD thesis, The Austrailian National University, 1999.

[49] J. D. Tygar, A. Gupta, O. Shmueli, and J. Widom. Atomicity versus anonymity: Distributed transactions for electronic commerce. In *Proceedings of the 24th Annual International Conference on Very Large Data Bases*, pages 1–12, New York, NY., August 1998.

[50] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 93–109, Kiawah Island Resort, SC, December 1999.

[51] A. Westerlund and J. Danielsson. Arla—afree afs client. In *Proceedings of 1998 USENIX, Freenix track*, New Orleans, LA, June 1998.

Secure routing for structured peer-to-peer overlay networks

Miguel Castro[1], Peter Druschel[2], Ayalvadi Ganesh[1], Antony Rowstron[1] and Dan S. Wallach[2]

[1]*Microsoft Research Ltd., 7 J J Thomson Avenue, Cambridge, CB3 0FB, UK*
{mcastro,ajg,antr}@microsoft.com
[2]*Rice University, 6100 Main Street, MS 132, Houston, TX 77005-1892, USA*
{druschel,dwallach}@cs.rice.edu

Abstract

Structured peer-to-peer overlay networks provide a substrate for the construction of large-scale, decentralized applications, including distributed storage, group communication, and content distribution. These overlays are highly resilient; they can route messages correctly even when a large fraction of the nodes crash or the network partitions. But current overlays are not secure; even a small fraction of malicious nodes can prevent correct message delivery throughout the overlay. This problem is particularly serious in open peer-to-peer systems, where many diverse, autonomous parties without preexisting trust relationships wish to pool their resources. This paper studies attacks aimed at preventing correct message delivery in structured peer-to-peer overlays and presents defenses to these attacks. We describe and evaluate techniques that allow nodes to join the overlay, to maintain routing state, and to forward messages securely in the presence of malicious nodes.

1 Introduction

Structured peer-to-peer (p2p) overlays like CAN [16], Chord [20], Pastry [17] and Tapestry [21] provide a self-organizing substrate for large-scale peer-to-peer applications. These systems provide a powerful platform for the construction of a variety of decentralized services, including network storage, content distribution, and application-level multicast. Structured overlays allow applications to locate any object in a probabilistically bounded, small number of network hops, while requiring per-node routing tables with only a small number of entries. Moreover, the systems are scalable, fault-tolerant and provide effective load balancing.

However, to fully realize the potential of the p2p paradigm, such overlay networks must be able to support an open environment where mutually distrusting parties with conflicting interests are allowed to join. Even in a closed system of sufficiently large scale, it may be unrealistic to assume that none of the participating nodes have been compromised by attackers. Thus, structured overlays must be robust to a variety of security attacks, including the case where a fraction of the participating nodes act maliciously. Such nodes may mis-route, corrupt, or drop messages and routing information. Additionally, they may attempt to assume the identity of other nodes and corrupt or delete objects they are supposed to store on behalf of the system.

In this paper, we consider security issues in structured p2p overlay networks. We describe attacks that can be mounted against such overlays and the applications they support, and present the design of secure techniques that can thwart such attacks. In particular, we identify *secure routing* as a key building block that can be combined with existing, application-specific security techniques to construct secure, decentralized applications upon structured overlays. Secure routing requires (1) a secure assignment of node identifiers, (2) secure routing table maintenance, and (3) secure message forwarding. We present techniques for each of these problems, and show how using these techniques, secure routing can be maintained efficiently despite up to 25% of malicious participating nodes. Moreover, we show that the overhead of secure routing is acceptable and proportional to the fraction of malicious nodes.

The rest of this paper is organized as follows. Section 2 gives some background on structured p2p overlays, specifies models and assumptions, and defines secure routing. Sections 3, 4 and 5 present attacks on and solutions for assignment of identifiers to nodes, routing table maintenance and message forwarding, respectively. Section 6 explains how the overhead of secure routing can be minimized by using self-certifying data. Finally, Section 7 discusses related work and Section 8 provides conclusions.

2 Background, models and solution

In this section, we present some background on structured p2p overlay protocols like CAN, Chord, Tapestry and Pastry. Space limitations prevent us from giving a detailed overview of each protocol. Instead, we describe an abstract model of structured p2p overlay networks that we use to keep the discussion independent of any particular protocol. For concreteness, we also give an overview of Pastry and point out relevant differences with the other protocols. Next, we describe models and assumptions used later in the paper about how faulty nodes may behave. Finally, we define secure routing and outline our solution.

Throughout this paper, most of the analyses and techniques are presented in terms of our abstract model, and should apply to other structured overlays except when otherwise noted. However, the security and performance of our techniques was fully evaluated only in the context of Pastry; a full evaluation of the techniques in other protocols is future work.

2.1 Routing overlay model

We define an abstract model of a structured p2p routing overlay, designed to capture the key concepts common to overlays like CAN, Chord, Tapestry and Pastry.

In our model, participating nodes are assigned uniform random identifiers, *nodeIds*, from a large *id space* (e.g., the set of 128-bit unsigned integers). Application-specific objects are assigned unique identifiers, called *keys*, selected from the same id space. Each key is mapped by the overlay to a unique live node, called the key's *root*. The protocol routes messages with a given key to its associated root.

To route messages efficiently, each node maintains a *routing table* with nodeIds of other nodes and their associated IP addresses. Moreover, each node maintains a *neighbor set*, consisting of some number of nodes with nodeIds near the current node in the id space. Since nodeId assignment is random, any neighbor set represents a random sample of all participating nodes.

For fault tolerance, application objects are stored at more than one node in the overlay. A *replica function* maps an object's key to a set of *replica keys*, such that the set of *replica roots* associated with the replica keys represents a random sample of participating nodes in the overlay. Each replica root stores a copy of the object.

Next, we discuss existing structured p2p overlay protocols and how they relate to our abstract model.

2.2 Pastry

Pastry nodeIds are assigned randomly with uniform distribution from a circular 128-bit id space. Given a 128-bit key, Pastry routes an associated message toward the live node whose nodeId is numerically closest to the key. Each Pastry node keeps track of its neighbor set and notifies applications of changes in the set.

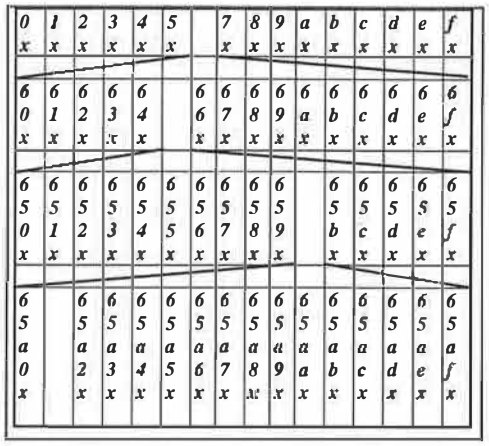
Node state: For the purpose of routing, nodeIds and keys are thought of as a sequence of digits in base 2^b (b is a configuration parameter with typical value 4). A node's routing table is organized into $128/2^b$ rows and 2^b columns. The 2^b entries in row r of the routing table contain the IP addresses of nodes whose nodeIds share the first r digits with the present node's nodeId; the $r+1$th nodeId digit of the node in column c of row r equals c. The column in row r that corresponds to the value of the $r+1$th digit of the local node's nodeId remains empty. A routing table entry is left empty if no node with the appropriate nodeId prefix is known. Figure 1 depicts an example routing table.

Each node also maintains a neighbor set (called a "leaf set"). The leaf set is the set of l nodes with nodeIds that are numerically closest to the present node's nodeId, with $l/2$ larger and $l/2$ smaller nodeIds than the current node's id. The value of l is constant for all nodes in the overlay, with a typical value of approximately $\lceil 8 * log_{2^b}N \rceil$, where N is the number of expected nodes in the overlay. The leaf set ensures reliable message delivery and is used to store replicas of application objects.

Message routing: At each routing step, a node seeks to forward the message to a node in the routing table whose nodeId shares with the key a prefix that is at least one digit (or b bits) longer than the prefix that the key shares with the present node's id. If no such node can be found, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's id. If no appropriate node exists in either the routing table or neighbor set, then the current node or its immediate neighbor is the message's final destination.

Figure 2 shows the path of an example message. Analysis shows that the expected number of routing hops is slightly below $log_{2^b}N$, with a distribution that is tight around the mean. Moreover, simulation shows that the routing is highly resilient to crash failures.

To achieve self-organization, Pastry nodes must dynamically maintain their node state, i.e., the routing table and neighbor set, in the presence of node arrivals and node failures. A newly arriving node with the new nodeId X can initialize its state by asking any existing Pastry node A to route a special message using X as the key. The message is routed to the existing node Z with nodeId numerically closest to X. X then obtains the neighbor set from Z and constructs its routing table by copying rows from the routing tables of the nodes it encountered on the original route from A to Z. Finally, X announces its presence to the initial members of its neighbor set, which in turn update their own neighbor sets and routing tables. Similarly, the overlay can adapt to abrupt node failure by exchanging a small number of messages ($O(log_{2^b}N)$) among a small number of nodes.



0x	1x	2x	3x	4x	5x		7x	8x	9x	ax	bx	cx	dx	ex	fx
60x	61x	62x	63x	64x		66x	67x	68x	69x	6ax	6bx	6cx	6dx	6ex	6fx
650x	651x	652x	653x	654x	655x	656x	657x	658x	659x		65bx	65cx	65dx	65ex	65fx
65a0x		65a2x	65a3x	65a4x	65a5x	65a6x	65a7x	65a8x	65a9x	65aax	65abx	65acx	65adx	65aex	65afx

Figure 1: Routing table of a Pastry node with nodeId $65a1x$, $b = 4$. Digits are in base 16, x represents an arbitrary suffix.

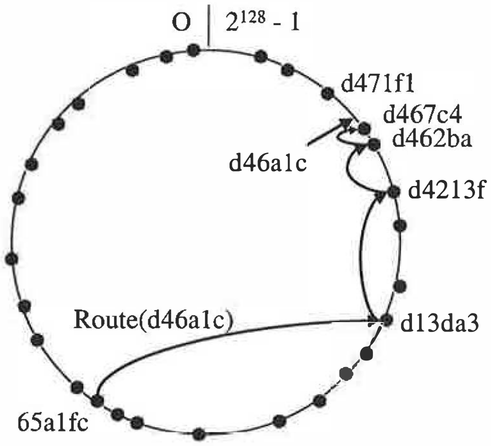


Figure 2: Routing a message from node $65a1fc$ with key $d46a1c$. The dots depict live nodes in Pastry's circular namespace.

2.3 CAN, Chord, Tapestry

Next, we briefly describe CAN, Chord and Tapestry, with an emphasis on the differences relative to Pastry.

Tapestry is very similar to Pastry but differs in its approach to mapping keys to nodes and in how it manages replication. In Tapestry, neighboring nodes in the namespace are not aware of each other. When a node's routing table does not have an entry for a node that matches a key's nth digit, the message is forwarded to the node with the next higher value in the nth digit, modulo 2^b, found in the routing table. This procedure, called *surrogate routing*, maps keys to a unique live node if the node routing tables are consistent. Tapestry does not have a direct analog to a neighbor set, although one can think of the lowest populated level of the Tapestry routing table as a neighbor set. For fault tolerance, Tapestry's replica function produces a set of random keys, yielding a set of replica roots at random points in the id space. The expected number of routing hops in Tapestry is $log_{2^b}N$.

Chord uses a 160-bit circular id space. Unlike Pastry, Chord forwards messages only in clockwise direction in the circular id space. Instead of the prefix-based routing table in Pastry, Chord nodes maintain a routing table consisting of up to 160 pointers to other live nodes (called a "finger table"). The ith entry in the finger table of node n refers to the live node with the smallest nodeId clockwise from $n + 2^{i-1}$. The first entry points to n's successor, and subsequent entries refer to nodes at repeatedly doubling distances from n. Each node in Chord also maintains pointers to its predecessor and to its n successors in the nodeId space (this successor list represents the neighbor set in our model). Like Pastry, Chord's replica function maps an object's key to the nodeIds in the neighbor set of the key's root, i.e., replicas are stored in the neighbor set of the key's root for fault tolerance. The expected number of routing hops in Chord is $\frac{1}{2}log_2N$.

CAN routes messages in a d-dimensional space, where each node maintains a routing table with $O(d)$ entries and any node can be reached in $(d/4)(N^{1/d})$ routing hops on average. The entries in a node's routing table refer to its neighbors in the d-dimensional space. CAN's neighbor table duals as both the routing table and the neighbor set in our model. Like Tapestry, CAN's replica function produces random keys for storing replicas at diverse locations. Unlike Pastry, Tapestry and Chord, CAN's routing table does not grow with the network size, but the number of routing hops grows faster than $logN$ in this case.

Tapestry and Pastry construct their overlay in a Internet topology-aware manner to reduce routing delays and network utilization. In these protocols, routing table entries can be chosen arbitrarily from an entire segment of the nodeId space without increasing the expected number of routing hops. The protocols exploit this by initializing the routing table to refer to nodes that are nearby in the network topology and have the appropriate nodeId prefix. This greatly facilitates proximity routing [17]. However, it also makes these systems vulnerable to certain attacks, as shown in Section 4.

The choice of entries in CAN's and Chord's routing tables is tightly constrained. The CAN routing table entries refer to specific neighboring nodes in each dimension, while the Chord finger table entries refer to specific points in the nodeId space. This makes proximity routing harder but it protects nodes from attacks that exploit attacking nodes' proximity to their victims.

2.4 System model

The system runs on a set of N nodes that form an overlay using one of the protocols described in the previous section. We assume a bound f $(0 \leq f < 1)$ on the fraction of nodes that may be faulty. Faults are modeled using a constrained-collusion Byzantine failure model, i.e., faulty nodes can behave arbitrarily and they may not all necessarily be operating as a single conspiracy. The set of faulty nodes is partitioned into independent coalitions, which are disjoint sets with size bounded by cN $(1/N \leq c \leq f)$. When $c = f$, all faulty nodes may collude with each other to cause the most damage to the system. We model the case where nodes are grouped into multiple independent coalitions by setting $c < f$. Members of a coalition can work together to corrupt the overlay but are unaware of nodes in other coalitions. We studied the behavior of the system with c ranging from $1/N$ to f to model different failure scenarios.

We assume that every node in the p2p overlay has a static IP address at which it can be contacted. In this paper, we ignore nodes with dynamically assigned IP addresses, and nodes behind network address translation boxes or firewalls. While p2p overlays can be extended to address these concerns, this paper focuses on more traditional network hosts.

The nodes communicate over normal Internet connections. We distinguish between two types of communication: *network-level*, where nodes communicate directly without routing through the overlay, and *overlay-level*, where messages are routed through the overlay using one of the protocols discussed in the previous section. We use cryptographic techniques to prevent adversaries from observing or modifying network-level communication between correct nodes. An adversary has complete control over network-level communication to and from nodes that it controls. This can compromise overlay-level communication that is routed through a faulty node. Adversaries may delay messages between correct nodes but we assume that any message sent by a correct node to a correct destination over an overlay route with no faulty nodes is delivered within time D with probability P_D.

2.5 Secure routing

Next, we define a secure routing primitive that can be combined with existing techniques to construct secure applications on structured p2p overlays. Subsequent sections show how to implement the secure routing primitive under the fault and network models that we described in the previous section.

The routing primitives implemented by current structured p2p overlays provide a best-effort service to deliver a message to a replica root associated with a given key. With malicious overlay nodes, the message may be dropped or corrupted, or it may be delivered to a malicious node instead of a legitimate replica root. Therefore, these primitives cannot be used to construct secure applications. For example, when inserting an object, an application cannot ensure that the replicas are placed on legitimate, diverse replica roots as opposed to faulty nodes that impersonate replica roots. Even if applications use cryptographic methods to authenticate objects, malicious nodes may still corrupt, delete, deny access to or supply stale copies of all replicas of an object.

To address this problem, we define a secure routing primitive. *The secure routing primitive ensures that when a non-faulty node sends a message to a key k, the message reaches all non-faulty members in the set of replica roots R_k with very high probability.* R_k is defined as the set of nodes that contains, for each member of the set of replica keys associated with k, a live root node that is responsible for that replica key. In Pastry, for instance, R_k is simply a set of live nodes with nodeIds numerically closest to the key. Secure routing ensures that (1) the message is eventually delivered, despite nodes that may corrupt, drop or misroute the message; and (2) the message is delivered to all legitimate replica roots for the key, despite nodes that may attempt to impersonate a replica root.

Secure routing can be combined with existing security techniques to safely maintain state in a structured p2p overlay. For instance, *self-certifying data* can be stored on the replica roots, or a Byzantine-fault-tolerant replication algorithm like BFT [4] can be used to maintain the replicated state. Secure routing guarantees that the replicas are initially placed on legitimate replica roots, and that a lookup message reaches a replica if one exists. Similarly, secure routing can be used to build other secure services, such as maintaining file metadata and user quotas in a distributed storage utility. The details of such services are beyond the scope of this paper.

Implementing the secure routing primitive requires the solution of three problems: securely assigning nodeIds to nodes, securely maintaining the routing tables, and securely forwarding messages. Secure nodeId assignment ensures that an attacker cannot choose the value of nodeIds assigned to the nodes that the attacker controls. Without it, the attacker could arrange to control all replicas of a given object, or to mediate all traffic to and from a victim node.

Secure routing table maintenance ensures that the fraction of faulty nodes that appear in the routing tables of correct nodes does not exceed, on average, the fraction of faulty nodes in the entire overlay. Without it, an attacker could prevent correct message delivery, given only a relatively small number of faulty nodes. Finally, secure message forwarding ensures that at least one copy of a message sent to a key reaches each correct replica root for the key with high probability. Sections 3, 4 and 5 describe solutions to each of these problems.

3 Secure nodeId assignment

The performance and security of structured p2p overlay networks depend on the fundamental assumption that there is a uniform random distribution of nodeIds that cannot be controlled by an attacker. This section discusses what goes wrong when the attacker violates this assumption, and how this problem can be addressed.

3.1 Attacks

Attackers who can choose nodeIds can compromise the integrity of a structured p2p overlay, without needing to control a particularly large fraction of the nodes. For example, an attacker may partition a Pastry or Chord overlay if she controls two complete and disjoint neighbor sets. Such attackers may also target particular victim nodes by carefully choosing nodeIds. For example, they may arrange for every entry in a victim's routing table and neighbor set to point to a hostile node in a Chord overlay. At that point, the victim's access to the overlay network is completely mediated by the attacker.

Attackers who can choose nodeIds can also control access to target objects. The attacker can choose the closest nodeIds to all replica keys for a particular target object, thus controlling all replica roots. As a result, the attacker could delete, corrupt, or deny access to the object. Even when attackers cannot choose nodeIds, they may still be able to mount all the attacks above (and more) if they can obtain a large number of legitimate nodeIds easily. This is known as a Sybil attack [10].

Previous approaches to nodeId assignment have either assumed nodeIds are chosen randomly by the new node [5] or compute nodeIds by hashing the IP address of the node [20]. Neither approach is secure because an attacker has the opportunity either to choose nodeIds that are not necessarily random, or to choose an IP address that hashes to a desired interval in the nodeId space. Particularly as IPv6 is deployed, even modest attackers will have more potential IP addresses at their disposal than there are likely to be nodes in a given p2p network.

3.2 Solution: certified nodeIds

One solution to securing the assignment of nodeIds is to delegate the problem to a central, trusted authority. We use a set of trusted certification authorities (CAs) to assign nodeIds to principals and to sign *nodeId certificates*, which bind a random nodeId to the public key that speaks for its principal and an IP address. The CAs ensure that nodeIds are chosen randomly from the id space, and prevent nodes from forging nodeIds. Furthermore, these certificates give the overlay a public key infrastructure, suitable for establishing encrypted and authenticated channels between nodes.

Like conventional CAs, ours can be offline to reduce the risk of exposing certificate signing keys. They are not involved in the regular operation of the overlay. Nodes with valid nodeId certificates can join the overlay, route messages, and leave repeatedly without involvement of the CAs. As with any CA infrastructure, the CA's public keys must be well known, and can be installed as part of the node software itself, as is done with current Web browsers.

The inclusion of an IP address in the certificate deserves some explanation. Some p2p protocols, such as Tapestry and Pastry, measure the network delay between nodes and choose routing table entries that minimize delay. If an attacker with multiple legitimate nodeId certificates could freely swap certificates among nodes it controls, it might be able to increase the fraction of attacker's nodes in a target node's routing table. By binding the nodeId to an IP address, it becomes harder for an attacker to move nodeIds across nodes. We allow multiple nodeId certificates per IP address because the IP addresses of nodes may change and because otherwise, attackers could deny service by hijacking victim's IP addresses.

A downside of binding nodeIds to IP addresses is that, if a node's IP address changes, either as a result of dynamic address assignment, host mobility, or organizational network changes, then the node's old certificate and nodeId become invalid. In p2p systems where IP addresses are allowed to change dynamically, nodeId swapping attacks may be unavoidable.

Certified nodeIds work well when nodes have fixed nodeIds, which is the case in Chord, Pastry, and Tapestry. However, it might be harder to secure nodeId assignment in CAN. CAN nodeIds represent a zone in a d-dimensional space that is split in half when a new node joins [16]. Both the nodeId of the original node and the nodeId of the joining node change during this process.

3.2.1 Sybil attacks

While nodeId assignment by a CA ensures that nodeIds are chosen randomly, it is also important to prevent an attacker from easily obtaining a large number of nodeId certificates. One solution is to require an attacker to pay money for certificates, via credit card or any other suitable mechanism. With this solution, the cost of an attack grows naturally with the size of the network. For example, if nodeId certificates cost \$20, controlling 10% of an overlay with 1,000 nodes costs \$2,000 and the cost rises to \$2,000,000 with 1,000,000 nodes. The cost of targeted attacks is even higher; it costs an expected \$20,000 to obtain the closest nodeId to a particular point in the id space in an overlay with 1,000 nodes. Apart from making attacks economically expensive, these fees can also fund the operation of the CAs.

Another solution is to bind nodeIds to real-world identities instead of charging money. In practice, different forms of CAs are suitable in different situations. The identity-based CA is the preferred solution in "virtual private" overlays run by an organization that already maintains employment or membership records with strong identity checks. In an open Internet deployment, a money-only CA may be more suitable because it avoids the complexities of authenticating real-world identities.

None of the known solutions to nodeId assignment are effective when the overlay network is very small. For small overlay networks, we must require that all members of the network are trusted not to cheat. Only when a network reaches a critical mass, where it becomes sufficiently hard for an attacker to muster enough resources to control a significant fraction of the overlay, should untrusted nodes be allowed to join.

3.3 Rejected: distributed nodeId generation

The CAs represent points of failure, vulnerable to both technical and legal attacks. Also, for some p2p networks, it may be cumbersome to require users to spend money or prove their real-world identities. Therefore, it would be desirable to construct secure p2p overlays without requiring centralized authorities, fees or identity checks. Unfortunately, fully decentralized nodeId assignment appears to have fundamental security limitations [10]. None of the methods we are aware of can ultimately prevent a determined attacker from acquiring a large collection of nodeIds.

However, several techniques may be able to, at a minimum, moderate the *rate* at which an attacker can acquire nodeIds. One possible solution is to require prospective nodes to solve crypto puzzles [15] to gain the right to use a nodeId, an approach that has been taken to address a number of denial of service attacks [13, 8]. Unfortunately, the cost of solving a crypto puzzle must be acceptable to the slowest legitimate node, yet the puzzle must be hard enough to sufficiently slow down an attacker with access to many fast machines. This conflict limits the effectiveness of any such technique.

For completeness, we briefly describe here one relatively simple approach to generate certified nodeIds in a completely distributed fashion using crypto puzzles. The idea is to require new nodes to generate a key pair with the property that the SHA-1 hash of the public key has the first p bits zero. The expected number of operations required to generate such a key pair is 2^p. The properties of public-key cryptography allow the nodes to use a secure hash of the public key as their nodeId. This hash should be computed using SHA-1 with a different initialization vector or MD5 to avoid reducing the number of random bits in nodeIds. Nodes can prove that they performed the required amount of work to use a nodeId without revealing information that would allow others to reuse their work. The value of p can be set to achieve the desired level of security.

It is also possible to bind IP addresses with nodeIds to avoid attacks on overlays that exploit network locality. The idea is to require nodes to consume resources in order to be able to use a given nodeId with an IP address. We do this by requiring nodes to find a string x such that SHA-1(SHA-1(*ipaddr,x*),*nodeId*) has p' bits equal to zero. Nodes would be required to present such an x for the pair (*nodeId,ipaddr*) to be accepted by others.

Finally, it is possible to periodically invalidate nodeIds by having some trusted entity broadcast to the overlay a message supplying a different initialization vector for the hash computations. This makes it harder for an attacker to accumulate many nodeIds over time and to reuse nodeIds computed for one overlay in another overlay. However, it requires legitimate nodes to periodically spend additional time and communication to maintain their membership in the overlay.

4 Secure routing table maintenance

We now turn our attention to the problem of secure routing table maintenance. The routing table maintenance mechanisms are used to create routing tables and neighbor sets for joining nodes, and to maintain them after creation. Ideally, each routing table and neighbor set should have an average fraction of only f random entries that point to nodes controlled by the attacker (called "bad entries"). But attackers can increase the fraction of bad entries by supplying bad routing updates, which reduces the probability of routing successfully to replica roots.

Preventing attackers from choosing nodeIds is necessary to avoid this problem but it is not sufficient as illustrated by the two attacks discussed next. We also discuss solutions to this problem.

4.1 Attacks

The first attack is aimed at routing algorithms that use network proximity information to improve routing efficiency: attackers may fake proximity to increase the fraction of bad routing table entries. For example, the network model that we assumed allows an attacker to control communication to and from faulty nodes that it controls. When a correct node p sends a probe to estimate delay to a faulty node with a certain nodeId, an attacker can intercept the probe and have the faulty node closest to p reply to it. If the attacker controls enough faulty nodes spread over the Internet, it can make nodes that it controls appear close to correct nodes to increase the probability that they are used for routing. The attack is harder when c (the maximal fraction of colluding nodes) is small even if f is large.

This attack can be ruled out by a more restrictive communication model, since nodeId certificates bind IP addresses to nodeIds (see Section 3.2). For example, if faulty nodes can only observe messages that are sent to their own IP address [19], this attack is prevented. But note that a rogue ISP or corporation with several offices around the world could easily perform this attack by configuring their routers appropriately. The attack is also possible if there is any other form of indirection that the attacker can control, e.g., mobile IPv6.

The second attack does not manipulate proximity information. Instead, it exploits the fact that it is hard to determine whether routing updates are legitimate in overlay protocols like Tapestry and Pastry. Nodes receive routing updates when they join the overlay and when other nodes join, and they fetch routing table rows from other nodes in their routing table periodically to patch holes and reduce hop delays. In these systems, attackers can more easily supply routing updates that always point to faulty nodes. This simple attack causes the fraction of bad routing table entries to increase toward one as the bad routing updates are propagated. More precisely, routing updates from correct nodes point to a faulty node with probability at least f whereas this probability can be as high as one for routing updates from faulty nodes. Correct nodes receive updates from other correct nodes with probability at most $1-f$ and from faulty nodes with probability at least f. Therefore, the probability that a routing table entry is faulty after an update is at least $(1-f)\times f + f \times 1$, which is greater than f. This effect cascades with each subsequent update, causing the fraction of faulty entries to tend towards one.

Systems without strong constraints on the set of nodeIds that can fill each routing table slot are more vulnerable to this attack. Pastry and Tapestry impose very weak constraints at the top levels of routing tables. This flexibility makes it hard to determine if routing updates are unbiased but it allows these systems to effectively exploit network proximity to improve routing performance. CAN and Chord impose strong constraints on nodeIds in routing table entries: they need to be the closest nodeIds to some point in the id space. This makes it hard to exploit network proximity to improve performance but it is good for security; if attackers cannot choose the nodeIds they control, the probability that an attacker controls the nodeId closest to a point in the id space is f.

4.2 Solution: constrained routing table

To enable secure routing table maintenance, it is important to impose strong constraints on the set of nodeIds that can fill each slot in a routing table. For example, the entry in each slot can be constrained to be the closest nodeId to some point in the id space as in Chord. This constraint can be verified and it is independent of network proximity information, which can be manipulated by attackers.

The solution that we propose uses two routing tables: one that exploits network proximity information for efficient routing (as in Pastry and Tapestry), and one that constrains routing table entries (as in Chord). In normal operation, the first routing table is used to forward messages to achieve good performance. The second one is used only when the efficient routing technique fails. We use the test in Section 5.2 to detect when routing fails.

We modified Pastry to use this solution. We use the normal locality-aware Pastry routing table and an additional *constrained* Pastry routing table. In the locality-aware routing table of a node with identifier i, the slot at level l and domain d can contain any nodeId that shares the first l digits with i and has the value d in the $l+1$st digit. In the constrained routing table, the entry is further constrained to point to the closest nodeId to a point p in the domain. We define p as follows: it shares the first l digits with i, it has the value d in the $l+1$st digit, and it has the same remaining digits as i.

Pastry's message forwarding works with the constrained routing table without modifications. The same would be true with Tapestry. But the algorithms to initialize and maintain the routing table were modified as follows.

All overlay routing algorithms rely on a *bootstrap node* to initialize the routing state of a newly joining node. The bootstrap node is responsible for routing a message using the nodeId of the joining node as the key. If the bootstrap node is faulty, it can completely corrupt the view of the overlay network as seen by the new node. Therefore, it is necessary to use a set of diverse bootstrap nodes large enough to ensure that with very high probability, at least one of them is correct. The use of nodeId certificates makes the task of choosing such a set easier because the attacker cannot forge nodeIds.

A newly joining node, n, picks a set of bootstrap nodes and asks all of them to route using its nodeId as the key. Then, non-faulty bootstrap nodes use secure forwarding techniques (described in Section 5.2) to obtain the neighbor set for the joining node. Node n collects the proposed neighbor sets from each of the bootstrap nodes, and picks the "closest" live nodeIds from each proposed set to be its neighbor set (where the definition of closest is protocol specific).

The locality-aware routing table is initialized as before by collecting rows from the nodes along the route to the nodeId. The difference is that there are several routes; n picks the entry with minimal network delay from the set of candidates it receives for each routing table slot.

Each entry in the constrained routing table can be initialized by using secure forwarding to obtain the live nodeId closest to the desired point p in the id space. This is similar to what is done in Chord. The problem is that it is quite expensive with $b > 1$ (recall that b controls the number of columns in the routing table of Tapestry and Pastry). To reduce the overhead, we can take advantage of the fact that, by induction, the constrained routing tables of the nodes in n's neighbor set point to entries that are close to the desired point p. Therefore, n can collect routing tables from the nodes in its neighbor set and use them to initialize its constrained routing table. From the set of candidates that it receives for each entry, it picks the nodeId that is closest to the desired point for that entry. As a side effect of this process, n informs the nodes in its neighbor set of its arrival.

We exploit the symmetry in the constrained routing table to inform nodes that need to update their routing tables to reflect n's arrival: n checks its neighbor set and the set of candidates for each entry to determine which candidates should update routing table entries to point to n. It informs those candidates of its arrival.

To ensure neighbor set stabilization in the absence of new joins and leaves, n informs the members of its neighbor set whenever it changes and it periodically retransmits this information until its receipt is acknowledged.

5 Secure message forwarding

The use of certified nodeIds and secure routing table maintenance ensure that each constrained routing table (and neighbor set) has an average fraction of only f random entries that point to nodes controlled by the attacker. But routing with the constrained routing table is not sufficient because the attacker can reduce the probability of successful delivery by simply not forwarding messages according to the algorithm. The attack is effective even when f is small, as we will show. This section describes an efficient solution to this problem.

5.1 Attacks

All structured p2p overlays provide a primitive to send a message to a key. In the absence of faults, the message is delivered to the root node for the key after an average of h routing hops. But routing may fail if any of the $h-1$ nodes along the route between the sender and the root are faulty; faulty nodes may simply drop the message, route the message to the wrong place, or pretend to be the key's root. Therefore, the probability of routing successfully between two correct nodes when a fraction f of the nodes is faulty is only: $(1-f)^{h-1}$, which is independent of c.

The root node for a key may itself be faulty. As discussed before, applications can tolerate root faults by replicating the information associated with the key on several nodes — the *replica roots*. Therefore, the probability of routing successfully to a correct replica root is only: $\sigma = (1-f)^h$. The value of h depends on the overlay: it is $(d/4)(N^{1/d})$ in CAN, $log_2(N)/2$ in Chord, and $log_{2^b}(N)$ in Pastry and Tapestry.

We ran simulations of Pastry to validate this model. The model predicts a probability of success slightly lower than the probability that we observed in the simulations (because the number of Pastry hops is slightly less than $log_{2^b}(N)$ on average [3]) but the error was below 2%.

Figure 3 plots the probability of routing to a correct replica in Pastry (computed using the model) for different values of f, N, and $b=4$. The probability drops quite fast when f or N increase. Even with only 10% of the nodes compromised, the probability of successful routing is only 65% when there are 100,000 nodes in a Pastry overlay.

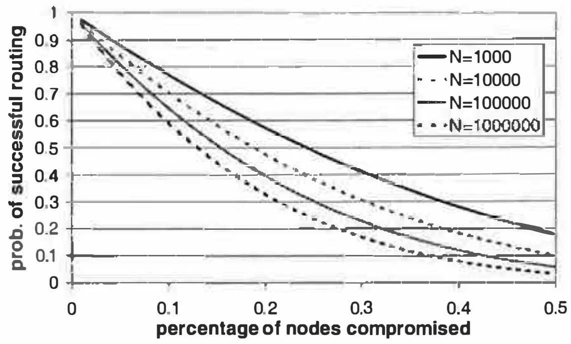


Figure 3: Probability of routing to a correct replica.

In CAN, Pastry, and Tapestry, applications can reduce the number of hops by increasing the value of d or b. Fewer hops increase the probability of routing correctly. For example, the probability of successful delivery with $f=0.1$ and 100,000 nodes is 65% in Pastry when $b=4$ and 75% when $b=6$. But increasing b also increases the cost of routing table maintenance; a high probability of routing success requires an impractically large value of b. Chord currently uses a fixed $b=1$, which results in a low probability of success, e.g., the probability is only 42% under the same conditions.

5.2 Solution: detect faults, use diverse routes

The results in Figure 3 show that it is important to devise mechanisms to route securely. We want a *secure routing primitive* that takes a message and a destination key and ensures that with very high probability at least one copy of the message reaches each correct replica root for the key. The question is how to do this efficiently.

Our approach is to route a message efficiently and to apply a *failure test* to determine if routing worked. We only use more expensive *redundant* routing when the failure test returns positive. In more detail, our secure routing primitive routes a message efficiently to the root of the destination key using the locality-aware routing table. Then, it collects the prospective set of replica roots from the prospective root node and applies the *failure test* to the set. If the test is negative, the prospective replica roots are accepted as the correct ones. If it is positive, message copies are sent over diverse routes toward the various replica roots such that with high probability each correct replica root is reached. We start by describing how to implement the failure test. Then we explain redundant routing and why we rejected an alternate approach called iterative routing.

5.2.1 Routing failure test

The failure test takes a key and a set of prospective replica roots for the key. It returns negative if the set of roots is likely to be correct for the key. Otherwise, it returns positive. Of course, routing can fail without the sender ever receiving a set of prospective replica roots. The sender detects this by starting a timer when it sends a message. If it does not receive a response before the timer expires, the failure test returns positive triggering the use of redundant routing.

Detecting routing failures is difficult because a coalition of faulty nodes can pretend to be the legitimate replica roots for a given key. Since the replica roots are determined by the structure of the overlay, a node whose nodeId is far from the key must rely on overlay routing to determine the correct set of replica roots. But if a message is routed by a faulty node, the adversary can fabricate a credible route and replica root set that contain only nodes it controls. Furthermore, it might be the case that the adversary just happens to legitimately control one of the actual replica roots. This problem is common to all structured p2p overlay protocols.

The routing failure test is based on the observation that the average density of nodeIds per unit of "volume" in the id space is greater than the average density of faulty nodeIds. The test works by comparing the density of nodeIds in the neighbor set of the sender with the density of nodeIds close to the replica roots of the destination key. We describe the test in detail only in the context of Pastry to simplify the presentation; the generalization to other overlays is straightforward. Overlays that distribute replica keys for a key uniformly over the id space can still use this check by comparing the density at the sender with the average distance between each replica key and its root's nodeId.

In Pastry, the set of replica roots for a key is a subset of the neighbor set of the key's root node, called the key's *root neigbor set*. Each correct node p computes the average numerical distance, μ_p, between consecutive nodeIds in its neighbor set. The neighbor set of p contains $l+1$ live nodes: p, the $l/2$ nodes with the closest nodeIds less than p's, and the $l/2$ nodes with the closest nodeIds greater than p's. To test a prospective root neighbor set, $rn = id_0, ..., id_{l+1}$, for a key x, p checks that:

1. all nodeIds in rn have a valid nodeId certificate, the closest nodeId to the key is the middle one, and the nodeIds satisfy the definition of a neighbor set
2. the average numerical distance, μ_{rn}, between consecutive nodeIds in rn satisfies: $\mu_{rn} < \mu_p \times \gamma$

If rn satisfies both conditions, the test returns negative; otherwise, it returns positive. The test can be inaccurate in one of two ways: it can return a *false positive* when the prospective root neighbor set is correct, or it can return a *false negative* when the prospective set is incorrect. We call the probability of false positives α and the probability of false negatives β. The parameter γ controls the trade off between α and β. Intuitively, increasing γ decreases α but it also increases β.

Assuming that there are N live nodes with nodeIds uniformly distributed over the id space (which has length $D = 2^{128}$), the distances between consecutive nodeIds are approximately independent exponential random variables with mean D/N for large N. The same holds for the distances between consecutive nodeIds of faulty nodes that can collude together but the mean is $D/(c \times N)$. It is interesting to note that α and β are independent of f. They only depend on the upper bound, c, on the fraction of colluding nodes because faulty nodes only know the identities of faulty nodes that they collude with.

Under these assumptions, we have derived the following expressions to compute α and β (see detailed derivation in the Appendix):

$$\alpha(n,k,\gamma) = \frac{n^n k^k e^{-n-k}}{(n-1)!(k-1)!} \int_0^\infty \frac{u^{n-1} e^{-n(u-1)}}{(n-1)!} \int_{\gamma u}^\infty \frac{v^{k-1} e^{-k(v-1)}}{(k-1)!} dvdu$$

$$\beta(n,k,\gamma,c) = \alpha(k,n,\frac{1}{\gamma c})$$

These expressions can be used to compute α and β numerically. We also computed the following closed-form upper bounds for α and β:

$$\alpha \le \exp\left\{-k\left[(r+1)\log\frac{r+\gamma}{r+1} - \log\gamma\right]\right\}$$

$$\beta \le \exp\left\{-k\left[(r+1)\log\frac{r+\gamma c}{r+1} + \log(\gamma c)\right]\right\}$$

where n is the number of distance samples used to compute μ_p, k is the number of distance samples used to compute μ_{rn}, and $r = n/k$. The test above used $n = k = l$.

The analysis shows that α and β are independent of N (provided $k \ll N$), and that the test's accuracy can be improved by increasing the number of distance samples used to compute the means. It is easy to increase the number of samples n used to compute μ_p by augmenting the mechanism that is already in place to stabilize neighbor sets. This mechanism propagates nodeIds that are added and removed from a neighbor set to the other members of the set; it can be extended to propagate nodeIds further but we omit the details due to lack of space. It is hard to increase the number of samples used to compute μ_{rn} because of some attacks that we describe below. Therefore, we keep $k = l$.

We ran several simulations to evaluate the effectiveness of our routing failure test. The simulations ran in a system with 100,000 random nodeIds. Figure 4 plots values of α and β for different values of γ with $f = c = 0.3$, the

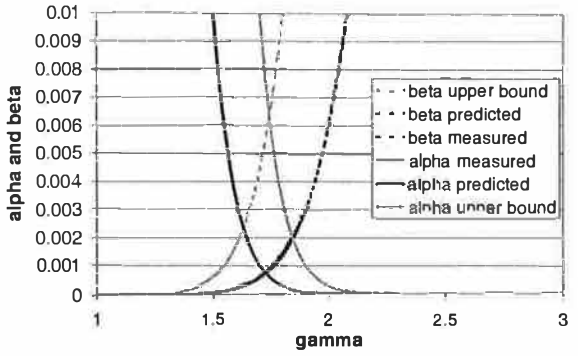


Figure 4: Routing failure test: probability of false positives (α) and negatives (β). The predicted curves are almost indistinguishable from the simulation measurements but the upper bounds are not tight.

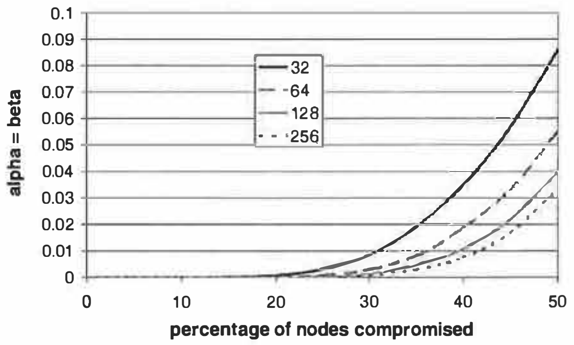


Figure 5: Routing failure test: minimum error probability without nodeId suppression attacks and varying number of samples.

number of samples at the sender is $n = 256$, and the number of root neighbors is $k = l = 32$. The figure shows predicted values computed numerically, the upper bounds, and values measured in the simulations. The predicted curves match the measured curves almost exactly but the upper bounds are not very tight. The minimum error is obtained when $\alpha = \beta$, which is equal to 0.0008 with $\gamma = 1.72$ in this case.

Attacks: There are several attacks that could invalidate the analysis and weaken our routing failure test. First, the attacker can collect nodeId certificates of nodes that have left the overlay, and use them to increase the density of a prospective root neighbor set. Second, the attacker can include both nodeIds of nodes it controls and nodeIds of correct nodes in a prospective root neighbor set. Both attacks can reduce the probability that messages reach all correct replica roots. The second attack is harder to counter in overlays that distribute replica keys over the id space because replica roots have no detailed knowledge about the nodeIds close to other replica keys.

These attacks can be avoided by having the sender contact all the prospective root neighbors to determine if they are live and if they have a nodeId certificate that was omitted from the prospective root neighbor set. To implement this efficiently, the prospective root returns to the sender a message with the list of nodeId certificates, a list with the secure hashes of the neighbor sets reported by each of the prospective root neighbors, and the set of nodeIds (not in the prospective root neighbor set) that are used to compute the hashes in this list. The sender checks that the hashes are consistent with the identifiers of the prospective root neighbors. Then, it sends each prospective root neigbor the corresponding neighbor set hash for confirmation.

In the absence of faults, the root neighbors will confirm the hashes and the sender can perform the density comparison immediately. For a sufficiently large timeout, this happens with probability $\tau = binom(0; k, f)$, where *binom* is the binomial distribution [6] and k is the number of root neighbors. With faulty nodes in the prospective root neighbor set, the routing failure test may require more communication before the density check can be run. We are still studying the best strategy to deal with this case. Currently, we consider the test failed when the prospective root neighbors don't agree and use redundant routing. But, it may be worthwhile investing some additional communication before reverting to redundant routing.

In addition to these attacks, there is a *nodeId suppression attack* that seems to be unavoidable and significantly decreases the accuracy of this test. The attacker can suppress nodeIds close to the sender by leaving the overlay, which increases β. Similarly, the attacker can suppress nodeIds in the root neighbor set, which increases α. Furthermore, the attacker can alternate between the two modes and honest nodes have no way of detecting in which mode they are operating.

We ran simulations to compute the minimum error probability ($\alpha = \beta$) with and without nodeId suppression attacks for different values of $c = f$. The probability of error increases fast with c and it is higher than 0.001 for $c \geq 0.35$ even with 256 samples at the sender. The nodeId suppression attack increases the minimum probability of error for large percentages of compromised nodes, e.g., the probability of error is higher than 0.001 for $c \geq 0.2$ even with 256 samples at the sender. Figures 5 and 6 show the results without and with nodeId suppression attacks, respectively.

These results indicate that our routing failure test is not very accurate. But, fortunately we can trade off an increase in α to achieve a target β and use redundant routing to disambiguate false positives. We ran simulations to determine the minimum α that can be achieved for a target $\beta = 0.001$ with different values of $c = f$, and different numbers of samples at the sender. Figure 7 shows

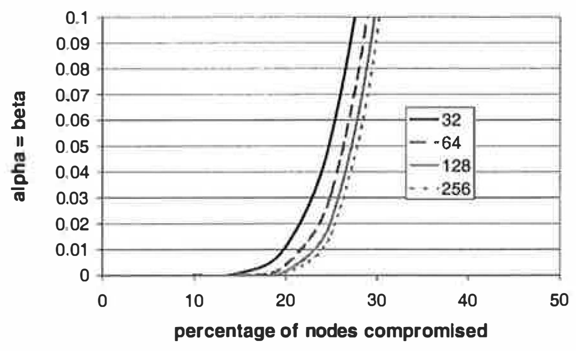


Figure 6: Routing failure test: minimum error probability with nodeId suppression attacks and varying number of samples.

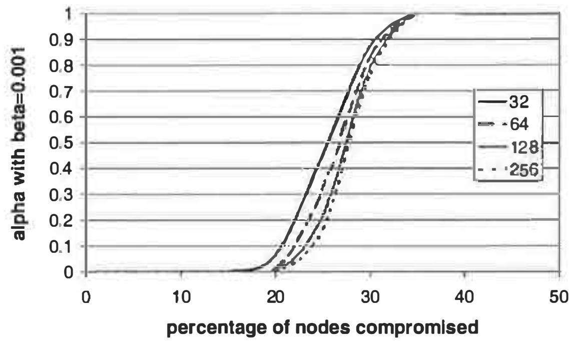


Figure 7: Routing failure test: probability of false positives for a false negative rate of 0.001 with nodeId suppression attacks and varying number of samples.

the results with nodeId suppression attacks.

The results show that the test is not meaningful for this target β and $c > 0.3$ with nodeId suppression attacks. However, setting $\gamma = 1.23$ with 256 samples at the sender enables the routing failure test to achieve the target β for $c \leq 0.3$. For this value of γ and with $c = 0.3$, nodeId suppression attacks can increase α to 0.77. But without nodeId suppression attacks the value of α is only 0.12, i.e., redundant routing is required 12% of the time.

5.2.2 Redundant routing

The redundant routing technique is invoked when the routing failure test is positive. The idea is simply to route copies of the message over multiple routes toward each of the destination key's replica roots. If enough copies of the message are sent along diverse routes to each replica key, all correct replica roots will receive at least one copy of the message with high probability.

The issue is how to ensure that routes are diverse. One approach is to ask the members of the sender's neighbor set to forward the copies of the message to the replica keys. This technique is sufficient in overlays that distribute the replica keys uniformly over the id space (e.g., CAN and Tapestry). But it is not sufficient in overlays that choose replica roots in the neighbor set of the key's root (e.g., Chord and Pastry) because the routes all converge on the key's root, which might be faulty. For these overlays, we developed a technique called *neighbor set anycast* that sends copies of the message toward the destination key until they reach a node with the key's root in its neighbor set. Then it uses the detailed knowledge that such a node has about the portion of the id space around the destination key to ensure that all correct replica roots receive a copy of the message.

To simplify the presentation, we only describe in detail how redundant routing works in Pastry. If a correct node p sends a message to a destination key x and the routing failure test is positive, it does the following:

(1) p sends r messages to the destination key x. Each message is forwarded via a different member of p's neighbor set; this causes the messages to use diverse routes. All messages are forwarded using the constrained routing table and they include a nonce.

(2) Any correct node that receives one of the messages and has x's root in its neighbor set returns its nodeId certificate and the nonce, signed with its private key, to p.

(3) p collects in a set $\mathcal{N}$ the $l/2+1$ nodeId certificates numerically closest to x on the left, and the $l/2+1$ closest to x on the right. Only certificates with valid signed nonces are added to $\mathcal{N}$ and they are first marked *pending*.

(4) After a timeout or after all r replies are received, p sends a list with the nodeIds in $\mathcal{N}$ to each node marked *pending* in $\mathcal{N}$ and marks the nodes *done*.

(5) Any correct node that receives this list forwards p's original message to the nodes in its neighbor set that are not in the list, or it sends a confirmation to p if there are no such nodes. This may cause steps 2 to 4 to be repeated.

(6) Once p has received a confirmation from each of the nodes in $\mathcal{N}$, or step 4 was executed three times, it computes the set of replica roots for x from $\mathcal{N}$.

If the timeout is sufficiently large and correct nodes have another correct node in each half of their neighbor set[1], the probability of reaching all correct replica roots of x is approximately equal to the probability that at least one of the anycast messages is forwarded over a route with no faults to a correct node with the key's root in its neighbor set. Assuming independent routes, this probability is:

$$1 - binom(0; r, (1-f)^{1+log_{2^b}N})$$

where *binom* is the binomial distribution [6] with 0 successful routes, r trials, and the probability of routing successfully in each trial is $(1-f)^{1+log_{2^b}N}$. The +1 counts

[1]The neighbor set size l should be chosen to ensure this with high probability

the extra hop for messages routed through a neighbor set member. The probability of success for this technique depends on f and is independent of c.

We also ran simulations to determine the probability of reaching all correct replica roots with our redundant routing technique. Figure 8 plots the predicted probability and the probability measured in the simulator for 100,000 nodes, $b = 4$, and $l = r = 32$. The analytic model matches the results well for high success probabilities. The results show that the probability of success is greater than 0.999 for $f < 0.3$. Therefore, this technique combined with our routing failure test can achieve a reliability of approximately 0.999 for $f < 0.3$.

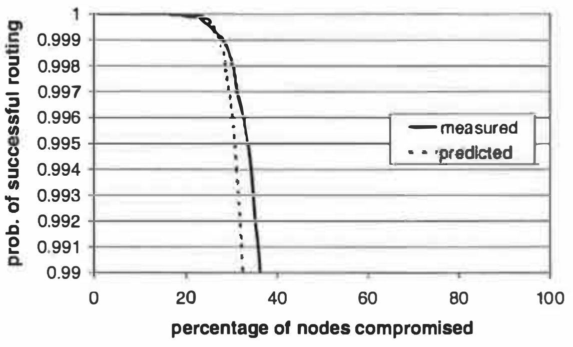


Figure 8: Model and simulation results for the probability of reaching all correct replica roots using redundant routing with neighbor set anycast.

We studied several versions of redundant routing that achieve the same probability of success but perform differently. For example, the signed nonces are used to ensure that the nodeId certificates in $\mathcal{N}$ belong to live nodes. But nodes can avoid signing nonces by periodically signing their clock reading in a system with loosely synchronized clocks, and no signatures are necessary if the attacker cannot forge IP source addresses. We are still exploring the design space. For example, it should be possible to improve performance significantly by sending the anycast messages one at a time and using a version of the routing failure test after each one. This approach would also work well when reading self-certifying data.

5.2.3 Putting it all together: performance

The performance of Pastry's secure routing primitive depends on the cost of the routing failure test, the cost of redundant routing, and on the probability that redundant routing can be avoided. This section presents an analysis of these costs and probability. For simplicity, we assume that all faulty nodes can collude ($c = f$), the number of probes used by redundant routing is equal to the neighbor set size ($r = l$), the number of samples at the source for routing failure tests is $n = 256$, and the number of nodes in the overlay is $N = 100{,}000$.

The cost of the routing failure test when it returns negative is an extra round-trip delay and $2l + 1$ messages. The total number of bytes in all messages is:

$$l \times (\text{IdSize} + 2\text{HashSize}) + (l + 1) \times \text{IdCertSize} + (2l + 1) \times \text{HdrSize}$$

Using PSS-R [1] for signing nodeId certificates with 1024-bit modulus and 512-bit modulus for the node public keys, the nodeId certificate size is 128B. Therefore, the extra bandwidth consumed by the routing failure test is approximately 5.6 KB with $l = 32$ and 2.9 KB with $l = 16$ (plus the space used up by message headers). When the test returns positive, it adds the same number of messages and bytes but the extra delay is the timeout period.

The cost of redundant routing depends on the value of f. The best case occurs when all of the root's neighbor set is added to $\mathcal{N}$ in the first iteration. In this case, redundant routing adds $\log_{2^b} N + 3$ extra message delays and $l \times (\log_{2^b} N + 3)$ messages. The total number of bytes in these messages is:

$$l \times (l \times \text{IdSize} + \text{IdCertSize} + \text{SigSize}) + l \times (\log_{2^b} N + 3) \times \text{HdrSize}$$

Using PSS-R for signing nonces, the signed nonce size is 64B. Therefore, the extra bandwidth consumed in this case is 22 KB with $l = 32$ and 7 KB with $l = 16$ (plus the space used up by message headers).

Under attack redundant routing adds a delay of at most three timeout periods and the expected number of extra messages is less than $l \times (\log_{2^b} N + 2) + (l - g) \times (3 + g)$, where $g = l \times (1 - f)^{\log_{2^b} N + 1}$ is the expected number of correct nodes in the root's neighbor set that is added to $\mathcal{N}$ in the first iteration. The expected number of messages is less than 451 with $l = 32$ and $f = 0.25$ and less than 188 with $l = 16$ and $f = 0.18$. The total number of bytes sent under attack is similar to the best case value except that the sender sends an additional $l(l - g) \times \text{IdSize}$ bytes in nodeId lists and the number of messages increases. This is an additional 12 KB with $l = 32$ and $f = 0.25$ and 1 KB with $l = 16$ and $f = 0.18$ (plus the space used up by message headers).

The probability of avoiding redundant routing is given by $\sigma \times \tau \times (1 - \alpha)$, where σ is the probability that the overlay routes the message to the correct replica root, τ is the probability that there are no faulty nodes in the neighbor set of the root, and α is the false positive rate of the routing failure test. We use $\sigma = (1 - f)^{\log_{2^b} N}$, which assumes that routing tables have an average of f random bad entries. This assumption holds for the locality-aware routing table in the absence of the attacks discussed in Section 4 and it holds for the constrained routing table even with these attacks. We do not have a good model of the effect of these attacks on the locality aware routing table but we believe that they are very hard to mount for small values of f (≤ 0.1).

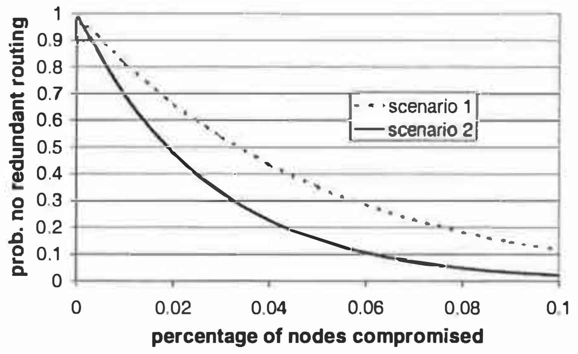


Figure 9: Probability of avoiding redundant routing in two scenarios: (1) $f \leq 0.18 \Rightarrow \Sigma \geq 0.999$ with $\gamma = 1.8$ and $l = 16$, and (2) $f \leq 0.25 \Rightarrow \Sigma \geq 0.999$ with $\gamma = 1.58$ and $l = 32$.

The parameters γ and l, should be set based on the desired security level, which can be expressed as the probability Σ that all correct replica roots receive a copy of the message. The overlay size and the assignment of values to the parameters implicitly define a bound on f. If this bound is exceeded, Σ will drop. For example, we saw that $f \leq 0.3 \Rightarrow \Sigma \geq 0.999$ with $\gamma = 1.23$ and $l = 32$. But redundant routing is invoked 12% of the time for this value of γ even with no faults.

One can trade off security for improved performance by increasing γ to reduce α, and by decreasing l to reduce the cost of the routing failure test and redundant routing and to increase τ. For example, consider the following two scenarios: (1)$f \leq 0.18 \Rightarrow \Sigma \geq 0.999$ with $\gamma = 1.8$ and $l = 16$, and (2) $f \leq 0.25 \Rightarrow \Sigma \geq 0.999$ with $\gamma = 1.58$ and $l = 32$. Figure 9 plots the probability of avoiding redundant routing in these two scenarios for different values of f. Without faults, redundant routing is invoked only 0.5% of the time in scenario (1) and 0.4% in (2). In the common case when the fraction of faulty nodes is small, the routing failure test improves performance significantly by avoiding the cost of redundant routing.

5.2.4 Rejected: checked iterative routing

An alternative to redundant routing is iterative routing, as suggested in Sit and Morris [19]: the sender starts by looking up the next hop in its routing table and setting a variable n to point to this node; then, the sender asks n for the next hop and updates n to point to the returned value. The process is repeated until this value is the root of the destination key.

Iterative routing doubles the cost relative to the more traditional recursive solution but it may increase the probability of routing successfully because it allows the sender to pick an alternative next hop when it fails to receive an entry from a node. This is not a strong defense against an attacker who can provide a faulty node as the next hop. However, iterative routing can be augmented with *hop tests* to check whether the next hop in a route is correct.

Hop tests are effective in systems like Chord or Pastry with the *constrained* routing table because each routing table entry should contain the nodeId closest to a specific point p in the id space. One can use a mechanism identical to the nodeId density checking that we used for the routing failure test. The only difference is that the average distance between consecutive nodeIds close to the sender is compared to the distance between the nodeId in the routing table entry and the desired point p. We ran simulations to compute the false positive and false negative rates for this approach with different values of c (these rates are independent of f). For example, the minimum error for this hop test ($\alpha = \beta$) is equal to approximately 0.35 with $c = 0.3$ and 256 samples to compute the mean at the sender.

The error is high because there is a single sample at the destination hop. However, our simulations indicate that iterative lookups using Pastry's constrained routing table with this hop check improve the probability of routing successfully. For example, the probability of routing successfully with $c = 0.3$, $N = 100{,}000$, $b = 4$, $l = 32$, and 256 samples to compute the mean at the sender, improves from below 0.3 to 0.4. But it adds an extra 2.7 hops to each route on average because of false positives.

We tried to increase the number of samples by having the sender fetch an entire routing table row during each iterative routing step without revealing the index of the required slot. Unfortunately, this performs worse than obtaining a single sample because the attacker can combine good and bad routing table entries to obtain a high average density.

We also tried to combine checked iterative routing with the redundant routing technique that we described before. We used checked iterative routing for the neighbor set anycast messages in the hope that the improved success probability for the iterative routes would result in an improvement over redundant routing with recursive routes. But there was no visible improvement, most likely because the iterative routes are less independent than the recursive routes. We conclude that the routing failure test combined with redundant routing is the most effective solution for implementing secure routing.

6 Self-certifying data

The secure routing primitive adds significant overhead over conventional routing. In this section, we describe how the use of secure routing can be minimized by using *self-certifying data*.

The reliance on secure routing can be reduced by storing self-certifying data in the overlay, i.e., data whose integrity can be verified by the client. This allows clients to use efficient routing to request a copy of an object.

If a client receives a copy of the object, it can check its integrity and resort to secure routing only when the integrity check fails or there was no response within a timeout period.

Self-certifying data does not help when inserting new objects in the overlay or when verifying that an object is not stored in the overlay. In these cases, we use the secure routing primitive to ensure that all correct replica roots are reached. Similarly, node joining requires secure routing. Nevertheless, self-certifying data can eliminate the overhead of secure routing in common cases.

Self-certifying data has been used in several systems. For example, CFS [7] uses a cryptographic hash of a file's contents as the key during insertion and lookup of the file, and PAST [18] inserts signed files into the overlay.

The technique can be extended to support mutable objects with strong consistency guarantees. One can use a system like PAST to store self-certifying *group descriptors* that identify the set of hosts responsible for replicating the object. Group descriptors can be used as follows. At object creation time, the owner of the object uses secure routing to insert a group descriptor into the overlay under a key that identifies the object. The descriptor contains the public keys and IP addresses of the object's replica holders and it is signed by the owner.

The replica group can run a Byzantine-fault-tolerant replication algorithm like BFT [4] and the initial group membership is the set of replica roots associated with the key. In this setting, read and write operations can be performed as follows: the client uses efficient routing to retrieve a group descriptor from the overlay and checks the descriptor's signature; if correct, it uses the information in the descriptor to authenticate the replica holders and to invoke a replicated operation. If the client fails to retrieve a valid descriptor or if it fails to authenticate the replica holders, it uses the secure routing primitive to obtain a correct group descriptor or to assert that the object does not exist. This procedure provides strong consistency guarantees (linearizability [11]) for reads and writes while avoiding the routing failure test in the common case.

Changing the membership of the group that is responsible for replicating an object is not trivial; it requires securely inserting a new group descriptor in the overlay and ensuring that clients can reliably detect stale group descriptors. The following technique allows groups to change membership while retaining strong consistency guarantees. Each group of hosts that stores replicas of a given object maintains a private/public key pair associated with the group. When the group membership changes, each host in the new membership generates a new key pair for the group, the hosts in the old membership use their old keys to sign a new group descriptor containing the new keys, and then delete the old keys. If this operation is performed by a quorum of replica holders before the bound on the number of faulty group members is exceeded [4], old replica holders that fail will not be able to collude to pretend they are the current group because they cannot form the quorum necessary to authenticate themselves to a client.

Group descriptors can be authenticated by following a signature chain that starts with an owner signature and has signatures of a quorum of replicas for each subsequent membership change. The chain can be shortened by a new signature from the owner or, alternatively, replicas can use proactive signature sharing [12] to avoid the need for chaining signatures.

7 Related work

Sit and Morris [19] present a framework for performing security analyses of p2p networks. Their adversarial model allows for nodes to generate packets with arbitrary contents, but assumes that nodes cannot intercept arbitrary traffic. They then present a taxonomy of possible attacks. At the routing layer, they identify node lookup, routing table maintenance, and network partitioning / virtualization as security risks. They also discuss issues in higher-level protocols, such as file storage, where nodes may not necessarily maintain the necessary invariants, such as storage replication. Finally, they discuss various classes of denial-of-service attacks, including rapidly joining and leaving the network, or arranging for other nodes to send bulk volumes of data to overload a victim's network connection (i.e., distributed denial of service attacks).

Dingledine *et al.* [9] and Douceur [10] discuss address spoofing attacks. With a large number of potentially malicious nodes in the system and without a trusted central authority to certify node identities, it becomes very difficult to know whether you can trust the claimed identity of somebody to whom you have never before communicated. Dingledine proposes to address this with various schemes, including the use of micro-cash, that allow nodes to build up *reputations*.

Bellovin [2] identifies a number of issues with Napster and Gnutella. He discusses how difficult it might be to limit Napster and Gnutella use via firewalls, and how they can leak information that users might consider private, such as the search queries they issue to the network. Bellovin also expresses concern over Gnutella's "push" feature, intended to work around firewalls, which might be useful for distributed denial of service attacks. He considers Napster's centralized architecture to be more secure against such attacks, although it requires all users to trust the central server.

It is worthwhile mentioning a very elegant alternative solution for secure routing table maintenance and forwarding that we rejected. This solution replaces each node by a group of diverse replicas as suggested by Lynch *et al.* [14]. The replicas are coordinated using a state machine replication algorithm like BFT [4] that can tolerate Byzantine faults. BFT can replicate arbitrary state machines and, therefore, it can replicate Pastry's routing table maintenance and forwarding protocols. Additionally, the algorithm in [14] provides strong consistency guarantees for overlay routing and maintenance.

However, there are two disadvantages: the solution is expensive even without faults, and it is less resilient than the solution that we propose. Each routing step is expensive because it requires an agreement protocol between the replicas. Since the replicas should be geographically dispersed to reduce the probability of correlated faults, agreement latency will be high. Additionally, each group of replicas must have less than $1/3$ of its nodes faulty. This bound on the number of faulty replicas per group results in a relatively low probability of successful routing. The probability that a replica group with r replicas is correct when a fraction f of the nodes in the Pastry overlay is compromised is $\sum_{i=0}^{\lfloor r/3 \rfloor} binom(i;r,f)$, where *binom* denotes the binomial distribution with i successes, r trials, and probability of success f. For example, the probability that a replica group is correct with 20% of the nodes compromised and 32 replicas is less than 93%. In this example, the probability of routing correctly with 100,000 nodes in the overlay is only 72%.

8 Conclusions

Structured peer-to-peer overlay networks have previously assumed a fail-stop model for nodes; any node accessible in the network was assumed to correctly follow the protocol. However, if nodes are malicious and conspire with each other, it is possible for a small number of nodes to compromise the overlay and the applications built upon it. This paper has presented the design and analysis of techniques for secure node joining, routing table maintenance, and message forwarding in structured p2p overlays. These techniques provide secure routing, which can be combined with existing techniques to construct applications that are robust in the presence of malicious participants. A routing failure test allows the use of efficient proximity-aware routing in the common case, resorting to the more costly redundant routing technique only when the test indicates possible interference by an attacker. Moreover, we show how the use of secure routing can be reduced by using self-certifying application data. These techniques allow us to tolerate up to 25% malicious nodes while providing good performance when the fraction of compromised nodes is small.

Acknowledgments

We wish to thank Robert Morris, Rodrigo Rodrigues, Fabien Petitcolas, our shepherd David Wetherall and the anonymous referees for their helpful comments. We also wish to thank Adam Stubblefield for many discussions on the nodeId assignment problem. This work was supported in part by grants from Texas ATP (003604-0079-2001) and NSF (CCR-9985332).

References

[1] M. Bellare and P. Rogaway. The exact security of digital signatures- How to sign with RSA and Rabin. In *Advances in Cryptology - EUROCRYPT 96, Lecture Notes in Computer Science, Vol. 1070*. Springer-Verlag, 1996.

[2] Steve Bellovin. Security aspects of Napster and Gnutella. In *2001 Usenix Annual Technical Conference*, Boston, Massachusetts, June 2001. Invited talk.

[3] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, May 2002.

[4] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, Louisiana, February 1999.

[5] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, July 2000. ICSI, Berkeley, California.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1990.

[7] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP'01*, Banff, Canada, October 2001.

[8] Drew Dean and Adam Stubblefield. Using client puzzles to protect TLS. In *10th Usenix Security Symposium*, pages 1–8, Washington, D.C., August 2001.

[9] Roger Dingledine, Michael J. Freedman, and David Molnar. Accountability measures for peer-to-peer systems. In *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly and Associates, November 2000.

[10] John R. Douceur. The Sybil attack. In *Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, Massachusetts, March 2002.

[11] M. P. Herlihy and J. M. Wing. Axioms for Concurrent Objects. In *Proceedings of 14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[12] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *Proc. of the 1997 ACM Conference on Computers and Communication Security*, 1997.

[13] Ari Juels and John Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. In *Internet Society Symposium on Network and Distributed System Security (NDSS '99)*, pages 151–165, San Diego, California, February 1999.

[14] Nancy Lynch, Dahlia Malkhi, and David Ratajczak. Atomic data access in content addressable networks. In *Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, Massachusetts, March 2002.

[15] Ralph C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, April 1978.

[16] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM'01*, San Diego, California, August 2001.

[17] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, November 2001.

[18] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. ACM SOSP'01*, Banff, Canada, October 2001.

[19] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, Massachusetts, March 2002.

[20] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, San Diego, California, August 2001.

[21] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

Appendix

This appendix describes an analytic model for the probability of false positives and negatives in the routing failure test.

We assume that there exist N nodeIds distributed uniformly at random on an interval of length $D = 2^{128}$. If N is large and we look at the K nodeIds closest to an arbitrarily chosen location on this interval (for some $K \ll N$), the location of these K nodeIds is well approximated in distribution by a Poisson process of rate N/D. In particular, the inter-point distances are approximately independent exponential random variables with mean D/N.

Let F_1 denote the exponential distribution with mean $\mu_1 = D/N$ and F_2 the exponential distribution with mean $\mu_2 = D/Nf$, where f is the fraction of faulty nodes. Suppose $y_1, \ldots, y_k$ are independent identically distributed (iid) and are drawn from one of these two distributions and we are required to identify which distribution they are drawn from, e.g., $y_1, \ldots, y_k$ can be a prospective set of replica roots in Pastry and we are trying to determine if the set is correct or if it contains only faulty nodes. An optimal hypothesis test is based on comparing the likelihood ratio to a threshold; by writing down the likelihood ratio, we see that this is equivalent to comparing the sample mean, denoted μ_y, to a threshold T.

We are in a situation where N is unknown but we have samples $x_1, \ldots, x_n$ from F_1 (i.e., the samples that we collect from the nodeIds close to the sender in the id space). We propose the following hypothesis test: choose a threshold of the form $\gamma\mu_x$, for some constant $\gamma \in (1, 1/f)$, and accept/reject the hypothesis that Y_i are iid F_1 by comparing μ_y to this threshold. We now compute the false positive probability, α, and the false negative probability, β, for this test.

Denote n/k by r and assume without loss of generality that r is an integer. For $i = 1, \ldots, k$, define

$$Z_i = Y_i - \frac{\gamma}{r}(X_{(i-1)r+1} + \ldots + X_{ir}),$$

and note that the Z_i are iid random variables. Let S_j denote the sum of j iid exponential random variables with mean $\mu_1 = D/N$. The event that $\mu_Y > \gamma\mu_X$ is then the event that $\sum_{i=1}^k Z_i > 0$. Thus,

$$\alpha(n,k,\gamma) = P_1\left(\sum_{i=1}^{k} Z_i > 0\right) = P\left(\frac{1}{k}S_k > \frac{\gamma}{n}S_n\right), \qquad (1)$$

where we write P_1 to denote probabilities when the Y_i have distribution F_1. Recalling that S_j has the gamma distribution with shape parameter j and scale parameter $1/\mu_1$, we can rewrite the above as

$$\alpha(n,k,\gamma) = \int_0^\infty \frac{(x/\mu_1)^{n-1}}{\mu_1(n-1)!}e^{-\mu_1 x}\int_{\frac{\gamma k}{n}}^\infty \frac{(x/\mu_1)^{k-1}}{\mu_1(k-1)!}e^{-\mu_1 y}dydx$$

$$= \frac{n^n k^k e^{-n-k}}{(n-1)!(k-1)!}\int_0^\infty \frac{u^{n-1}e^{-n(u-1)}}{(n-1)!}\int_{\gamma u}^\infty \frac{v^{k-1}e^{-k(v-1)}}{(k-1)!}dvdu$$

where we used the change of variables $u = x/(n\mu_1)$ and $v = y/(k\mu_1)$ to obtain the last equality. This expression can be used to compute α numerically.

We now derive a simple closed-form expression for an upper bound on α. The bound shows that α decays exponentially in the sample size, k, and in fact captures the exact exponential rate of decay. For arbitrary $\theta \geq 0$, we have by Chernoff's bound that

$$\alpha \leq E[\exp(\theta\sum_{i=1}^{k} Z_i)] = \left(E[e^{\theta Y_1}]\right)^k \left(E[\exp(-\frac{\gamma\theta}{r}X_1)]\right)^{rk}$$

Now, if X has an exponential distribution with mean μ, then $E[e^{\theta X}]$ is $1/(1-\theta\mu)$ for $\theta < 1/\mu$ and $+\infty$ for $\theta \geq 1/\mu$. Thus, for all $\theta \in [0, 1/\mu_1)$, we have

$$\log\alpha \leq -k\log(1-\theta\mu_1) - rk\log(1+\frac{\gamma\theta\mu_1}{r})$$

The tightest upper bound is obtained by minimising the expression on the right over $\theta \in [0, 1/\mu_1)$. The minimum is attained at $\theta = \frac{r}{r+1}\frac{\gamma-1}{\gamma\mu_1}$. Substituting this above yields the bound,

$$\alpha \leq \exp\left\{-k\left[(r+1)\log\frac{r+\gamma}{r+1} - \log\gamma\right]\right\} \qquad (2)$$

We can derive an expression for the false negative probability, β, along similar lines. Now, the Y_i are iid with distribution F_2, i.e., they are exponentially distributed with mean $\mu_2 = \mu_1/f$, and we are interested in the event that $\mu_Y \leq \gamma\mu_X$. If this happens, then we fail to reject the hypothesis that the Y_i have distribution F_1. Thus

$$\beta(n,k,\gamma,f) = P_2\left(\sum_{i=1}^{k} Z_i \leq 0\right),$$

where we write P_2 to denote probabilities when the Y_i are exponential with mean μ_1/f. In this case, Y_1 has the same distribution as X_1/f, so $\sum_{i=1}^k Y_i$ has the same distribution as $(\sum_{i=1}^k X_i)/f$, and we obtain using (1) that

$$\beta(n,k,\gamma,f) = P\left(\frac{1}{k}\frac{S_k^1}{f} < \frac{\gamma}{n}S_n^2\right) = P\left(\frac{1}{n}S_n^2 > \frac{1}{\gamma f}\frac{1}{k}S_k^1\right) = \alpha(k,n,\frac{1}{\gamma f})$$

This allows us to compute β numerically and by combining this with (2), we obtain the following closed-form upper bound

$$\beta \leq \exp\left\{-k\left[(r+1)\log\frac{r+\gamma f}{r+1} - \log(\gamma f)\right]\right\}$$

An Analysis of Internet Content Delivery Systems

Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy
Department of Computer Science & Engineering
University of Washington
{tzoompy,gummadi,rdunn,gribble,levy}@cs.washington.edu

Abstract

In the span of only a few years, the Internet has experienced an astronomical increase in the use of specialized content delivery systems, such as content delivery networks and peer-to-peer file sharing systems. Therefore, an understanding of content delivery on the Internet now requires a detailed understanding of how these systems are used in practice.

This paper examines content delivery from the point of view of four content delivery systems: HTTP web traffic, the Akamai content delivery network, and Kazaa and Gnutella peer-to-peer file sharing traffic. We collected a trace of all incoming and outgoing network traffic at the University of Washington, a large university with over 60,000 students, faculty, and staff. From this trace, we isolated and characterized traffic belonging to each of these four delivery classes. Our results (1) quantify the rapidly increasing importance of new content delivery systems, particularly peer-to-peer networks, (2) characterize the behavior of these systems from the perspectives of clients, objects, and servers, and (3) derive implications for caching in these systems.

1 Introduction

Few things compare with the growth of the Internet over the last decade, except perhaps its growth in the last several years. A key challenge for Internet infrastructure has been delivering increasingly complex data to a voracious and growing user population. The need to scale has led to the development of thousand-node clusters, global-scale content delivery networks, and more recently, self-managing peer-to-peer structures. These content delivery mechanisms are rapidly changing the nature of Internet content delivery and traffic; therefore, an understanding of the modern Internet requires a detailed understanding of these new mechanisms and the data they serve.

This paper examines content delivery by focusing on four content delivery systems: HTTP web traffic, the Akamai content delivery network, and the Kazaa and Gnutella peer-to-peer file sharing systems. To perform the study, we traced all incoming and outgoing Internet traffic at the University of Washington, a large university with over 60,000 students, faculty, and staff. For this paper, we analyze a nine day trace that saw over 500 million transactions and over 20 terabytes of HTTP data. From this data, we provide a detailed characterization and comparison of content delivery systems, and in particular, the latest peer-to-peer workloads. Our results quantify: (1) the extent to which peer-to-peer traffic has overwhelmed web traffic as a leading consumer of Internet bandwidth, (2) the dramatic differences in the characteristics of objects being transferred as a result, (3) the impact of the two-way nature of peer-to-peer communication, and (4) the ways in which peer-to-peer systems are *not* scaling, despite their explicitly scalable design. For example, our measurements show that an average peer of the Kazaa peer-to-peer network consumes 90 times more bandwidth than an average web client in our environment. Overall, we present important implications for large organizations, service providers, network infrastructure, and general content delivery.

The paper is organized as follows. Section 2 presents an overview of the content delivery systems examined in this paper, as well as related work. Section 3 describes the measurement methodology we used to collect and process our data. In Section 4 we give a high-level overview of the workload we have traced at the University of Washington. Section 5 provides a detailed analysis of our trace from the perspective of objects, clients, and servers, focusing in particular on a comparison of peer-to-peer and web traffic. Section 6 evaluates the potential for caching in content delivery networks and peer-to-peer networks, and Section 7 concludes and summarizes our results.

2 Overview of Content Delivery Systems

Three dominant content delivery systems exist today: the client/server oriented world-wide web, content delivery networks, and peer-to-peer file sharing systems. At a high level, these systems serve the same role of distributing content to users. However, the architectures of these systems differ significantly, and the differences affect their performance, their workloads, and the role caching can play. In this section, we present the architectures of these systems and describe previous studies of their behavior.

2.1 The World-Wide Web (WWW)

The basic architecture of the web is simple: using the HTTP [16] protocol, web clients running on users' machines request objects from web servers. Previous studies have examined many aspects of the web, including web workloads [2, 8, 15, 29], characterizing web objects [3, 11], and even modeling the hyperlink structure of the web [6, 21]. These studies suggest that most web objects are small (5-10KB), but the distribution of object sizes is heavy-tailed and very large objects exist. Web objects are accessed with a Zipf popularity distribution, as are web servers. The number of web objects is enormous (in the billions) and rapidly growing; most web objects are static, but an increasing number are generated dynamically.

The HTTP protocol includes provisions for consistency management. HTTP headers include caching pragmas that affect whether or not an object may be cached, and if so, for how long. Web caching helps to alleviate load on servers and backbone links, and can also serve to decrease object access latencies. Much research has focused on Web proxy caching [4, 5, 7, 11, 12] and, more recently, on coordinating state among multiple, cooperating proxy caches [13, 30, 33]; some of these proposals aim to create global caching structures [27, 34]. The results of these studies generally indicate that cache hit rates of 40-50% are achievable, but that hit rate increases only logarithmically with client population [36] and is constrained by the increasing amount of dynamically generated and hence uncacheable content.

2.2 Content Delivery Networks (CDNs)

Content delivery networks are dedicated collections of servers located strategically across the wide-area Internet. Content providers, such as web sites or streaming video sources, contract with commercial CDNs to host and distribute content. CDNs are compelling to content providers because the responsibility for hosting content is offloaded to the CDN infrastructure. Once in a CDN, content is replicated across the wide area, and hence is highly available. Since most CDNs have servers in ISP points of presence, clients can access topologically nearby replicas with low latency. The largest CDNs have thousands of servers dispersed throughout the Internet and are capable of sustaining large workloads and traffic hot-spots.

CDNs are tightly integrated into the existing web architecture, relying either on DNS interposition [19, 32] or on URL rewriting at origin servers to redirect HTTP requests to the nearest CDN replica. As with the web, the unit of transfer in a CDN is an object, and objects are named by URLs. Unlike the web, content providers need not manage web servers, since clients' requests are redirected to replicas hosted by the CDN. In practice, CDNs typically host static content such as images, advertisements, or media clips; content providers manage their own dynamic content, although dynamically generated web pages might contain embedded objects served by the CDN.

Previous research has investigated the use and effectiveness of content delivery networks [14], although the proprietary and closed nature of these systems tends to impede investigation. Two recent studies [22, 23] confirm that CDNs reduce average download response times, but that DNS redirection techniques add noticeable overhead because of DNS latencies. In another study [18], the authors argue that the true benefit of CDNs is that they help clients avoid the worst-case of badly performing replicas, rather than routing clients to a truly optimal replica. To the best of our knowledge, no study has yet compared the workloads of CDNs with other content delivery architectures.

2.3 Peer-to-Peer Systems (P2P)

Peer-to-peer file sharing systems have surged in popularity in recent years. In a P2P system, peers collaborate to form a distributed system for the purpose of exchanging content. Peers that connect to the system typically behave as servers as well as clients: a file that one peer downloads is often made available for upload to other peers. Participation is purely voluntary, and a recent study [31] has shown that most content-serving hosts are run by end-users, suffer from low availability, and have relatively low capacity network connections (modem, cable modems, or DSL routers).

Users interact with a P2P system in two ways: they attempt to locate objects of interest by issuing search queries, and once relevant objects have been located, users issue download requests for the content. Unlike the web and CDN systems, the primary mode of usage for P2P systems is a non-interactive, batch-style download of content.

P2P systems differ in how they provide search capabilities to clients [37]. Some systems, such as Napster [28], have large, logically centralized indexes maintained by a single company; peers automatically upload lists of available files to the central index, and queries are answered using this index. Other systems, such as Gnutella [10] and Freenet [9], broadcast search requests over an overlay network connecting the peers. More recent P2P systems, including Kazaa [20], use a hybrid architecture in which some peers are elected as "supernodes" in order to index content available at peers in a nearby neighborhood.

P2P systems also differ in how downloads proceed, once an object of interest has been located. Most systems transfer content over a direct connection between the object provider and the peer that issued the download request. A latency-improving optimization in some systems is to download multiple object fragments in parallel from multiple replicas. A recent study [24] has found the peer-to-peer traffic of a small ISP to be highly repetitive, showing great potential for caching.

3 Methodology

We use *passive network monitoring* to collect traces of traffic flowing between the University of Washington (UW) and the rest of the Internet. UW connects to its ISPs via two border routers; one router handles outbound traffic and the other inbound traffic. These two routers are fully connected to four switches on each of the four campus backbones. Each switch has a monitoring port that is used to send copies of the incoming and outgoing packets to our monitoring host.

Our tracing infrastructure is based on software developed by Wolman and Voelker for previous studies [35, 36]. We added several new components to identify, capture, and analyze Kazaa and Gnutella peer-to-peer traffic and Akamai CDN traffic. Overall, the tracing and analysis software is approximately 26,000 lines of code. Our monitoring host is a dual-processor Dell Precision Workstation 530 with 2.0 GHz Pentium III Xeon CPUs, a Gigabit Ethernet SysKonnect SK-9843 network card, and running FreeBSD 4.5.

Our software installs a kernel packet filter [26] to deliver TCP packets to a user-level process. This process reconstructs TCP flows, identifies HTTP requests within the flows (properly handling persistent HTTP connections), and extracts HTTP headers and other metadata from the flows. Because Kazaa and Gnutella use HTTP to exchange files, this infrastructure is able to capture P2P downloads as well as WWW and Akamai traffic. We anonymize sensitive information such as IP addresses and URLs, and log all extracted data to disk in a compressed binary representation.

3.1 Distinguishing Traffic Types

Our trace captures two types of traffic: *HTTP traffic*, which can be further broken down into WWW, Akamai, Kazaa, and Gnutella transfers, and *non-HTTP TCP traffic*, including Kazaa and Gnutella search traffic. If an HTTP request is directed to port 80, 8080, or 443 (SSL), we classify both the request and the associated response as WWW traffic. Similarly, we use ports 6346 and 6347 to identify Gnutella HTTP traffic, and port 1214 to identify Kazaa HTTP traffic. A small part of our captured HTTP traffic remains unidentifiable; we believe that most of this traffic can be attributed to less popular peer-to-peer systems (e.g., Napster [28]) and by compromised hosts turned into IRC or web servers on ports other than 80, 8080, or 444. For non-HTTP traffic, we use the same Gnutella and Kazaa ports to identify P2P search traffic.

Some WWW traffic is served by the Akamai content delivery network [1]. Akamai has deployed over 13,000 servers in more than 1,000 networks around the world [25]. We identify Akamai traffic as any HTTP traffic served by any Akamai server. To obtain a list of Akamai servers, we collected a list of 25,318 unique authoritative name servers, and sent a recursive DNS query to each server for a name in an Akamai-managed domain (e.g., `a388.g.akamaitech.net`). Because Akamai redirects DNS queries to nearby Akamai servers, we were able to collect a list of 3,966 unique Akamai servers in 928 different networks.

For the remainder of this paper, we will use the following definitions when classifying traffic:

- **Akamai:** HTTP traffic on port 80, 8080, or 443 that is served by an Akamai server.
- **WWW:** HTTP traffic on port 80, 8080, or 443 that is not served by an Akamai server; thus, for all of the analysis within this paper, "WWW traffic" does not include Akamai traffic.
- **Gnutella:** HTTP traffic sent to ports 6346 or 6347 (this includes file transfers, but excludes search and control traffic).
- **Kazaa:** HTTP traffic sent to port 1214 (this includes file transfers, but excludes search and control traffic).
- **P2P:** the union of Gnutella and Kazaa.
- **non-HTTP TCP traffic:** any other TCP traffic, including protocols such as NNTP and SMTP, HTTP traffic to ports other than those listed above, traffic from other P2P systems, and control or search traffic on Gnutella and Kazaa.

3.2 The Traceability of P2P Traffic

Gnutella is an overlay network over which search requests are flooded. Peers issuing search requests receive a list of other peers that have matching content. From this list, the peer that issued the request initiates a direct connection with one of the matching peers to download content. Because the Gnutella overlay is not structured to be efficient with respect to the physical network topology, most downloads initiated by UW peers connect to external hosts, and are therefore captured in our traces.

Although the details of Kazaa's architecture are proprietary, some elements are known. The Kazaa network is a two-level overlay: some well-connected peers serving as "supernodes" build indexes of the content stored on nearby "regular" peers. To find content, regular peers issue search requests to their supernodes. Supernodes appear to communicate amongst themselves to satisfy queries, returning locations of matching objects to the requesting peer. Kazaa appears to direct peers to nearby objects, although the details of how this is done, or how successful the system is at doing it, are not known.

To download an object, a peer initiates one or more connections to other peers that have replicas of the object. The downloading peer may transfer the entire object in one connection from a single peer, or it may choose to download multiple fragments in parallel from multiple peers.

	WWW		Akamai		Kazaa		Gnutella	
	inbound	outbound	inbound	outbound	inbound	outbound	inbound	outbound
HTTP transactions	329,072,253	73,001,891	33,486,508	N/A	11,140,861	19,190,902	1,576,048	1,321,999
unique objects	72,818,997	3,412,647	1,558,852	N/A	111,437	166,442	5,274	2,092
clients	39,285	1,231,308	34,801	N/A	4,644	611,005	2,151	25,336
servers	403,087	9,821	350	N/A	281,026	3,888	20,582	412
bytes transferred	1.51 TB	3.02 TB	64.79 GB	N/A	1.78 TB	13.57 TB	28.76 GB	60.38 GB
median object size	1,976 B	4,646 B	2,001 B	N/A	3.75 MB	3.67 MB	4.26 MB	4.08 MB
mean object size	24,687 B	82,385 B	12,936 B	N/A	27.78 MB	19.07 MB	19.16 MB	9.78 MB

Table 1. HTTP trace summary statistics: trace statistics, broken down by content delivery system; ***inbound*** refers to transfers from Internet servers to UW clients, and *outbound* refers to transfers from UW servers to Internet clients. Our trace was collected over a nine day period, from Tuesday May 28th through Thursday June 6th, 2002.

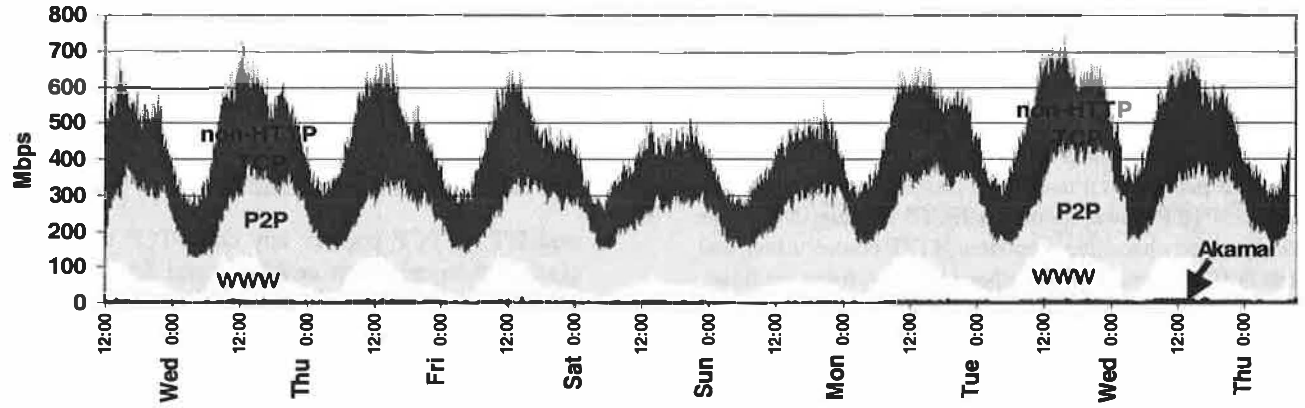


Figure 1. TCP bandwidth: total TCP bandwidth consumed by HTTP transfers for different content delivery systems. Each band is cumulative; this means that at noon on the first Wednesday, Akamai consumed approximately 10 Mbps, WWW consumed approximately 100 Mbps, P2P consumed approximately 200 Mbps, and non-HTTP TCP consumed approximately 300 Mbps, for a total of 610 Mbps.

The ability for a Kazaa peer to download an object in fragments complicates our trace. Download requests from external peers seen in our trace are often for fragments rather than entire objects.

4 High-Level Data Characteristics

This section presents a high-level characterization of our trace data. Table 1 shows summary statistics of object transfers. This table separates statistics from the four content delivery systems, and further separates inbound data (data requested by UW clients from outside servers) from outbound data (data requested by external clients from UW servers).

Despite its large client population, the University is a net *provider* rather than consumer of HTTP data, exporting 16.65 TB but importing only 3.44 TB. The peer-to-peer systems, and Kazaa in particular, account for a large percentage of the bytes exported and the total bytes transferred, despite their much smaller internal and external client populations. Much of this is attributable to a large difference in average object sizes between WWW and P2P systems.

The number of clients and servers in Table 1 shows the extent of participation in these systems. For the web, 39,285 UW clients accessed 403,437 Internet web servers, while for Kazaa, 4,644 UW clients accessed 281,026 external Internet servers. For Akamai, 34,801 UW clients download Akamai-hosted content provided by 350 different Akamai servers. In the reverse direction, 1,231,308 Internet clients accessed UW web content, while 611,005 clients accessed UW-hosted Kazaa content.

Figure 1 shows the total TCP bandwidth consumed in both directions over the trace period. The shaded areas show HTTP traffic, broken down by content delivery system; Kazaa and Gnutella traffic are grouped together under the label "P2P." All systems show a typical diurnal cycle. The smallest bandwidth consumer is Akamai, which currently constitutes only 0.2% of observed TCP traffic. Gnutella consumes 6.04%, and WWW traffic is the next largest, consuming 14.3% of TCP traffic. Kazaa is currently the largest contributor, consuming 36.9% of TCP bytes. These four content delivery systems account for 57% of total TCP traffic, leaving 43% for other TCP-based network protocols (streaming media, news, mail, and so on). TCP traffic represents over 97% of all network traffic at UW. This closely matches published data on Internet 2 usage [17].

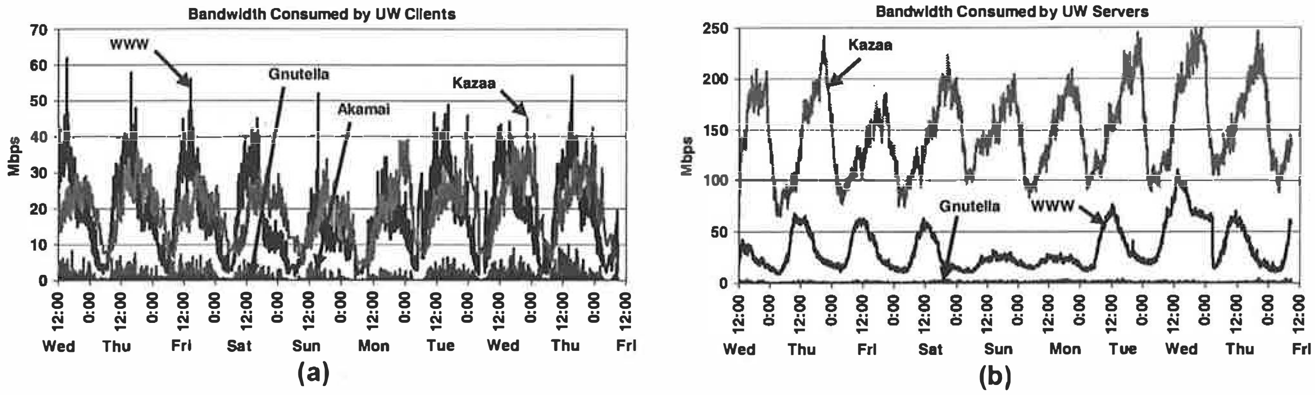


Figure 2. **UW client and server TCP bandwidth:** bandwidth over time (a) accountable to web and P2P downloads from UW clients, and (b) accountable to web and P2P uploads from UW servers.

Figures 2a and 2b show inbound and outbound data bandwidths, respectively. From Figure 2a we see that while both WWW and Kazaa have diurnal cycles, the cycles are offset in time, with WWW peaking in the middle of the day and Kazaa peaking late at night. For UW-initiated requests, WWW and Kazaa peak bandwidths have the same order of magnitude; however, for requests from external clients to UW servers, the peak Kazaa bandwidth dominates WWW by a factor of three. Note that the Y-axis scales of the graphs are different; WWW peak bandwidth is approximately the same in both directions, while external Kazaa clients consume 7.6 times more bandwidth than UW Kazaa clients.

Figure 3a and 3b show the top 10 content types requested by UW clients, ordered by bytes downloaded and number of downloads. While GIF and JPEG images account for 42% of requests, they account for only 16.3% of the bytes transferred. On the other hand, AVI and MPG videos, which account for 29.3% of the bytes transferred, constitute only 0.41% of requests. HTML is significant, accounting for 14.6% of bytes and 17.8% of requests. The 9.9% of bytes labelled "HASHED" in Figure 3a are Kazaa transfers that cannot be identified; of the non-hashed Kazaa traffic that can be identified, AVI and MPG account for 79% of the bytes, while 13.6% of the bytes are MP3.

It is interesting to compare these figures with corresponding measurements from our 1999 study of the same population [35]. Looking at bytes transferred as a percent of total HTTP traffic, HTML traffic has decreased 43% and GIF/JPG has decreased 59%. At the same time, AVI/MPG (and Quicktime) traffic has increased by nearly 400%, while MP3 traffic has increased by nearly 300%. (These percentages numbers include an estimate of the appropriate portion of the hashed bytes contributing to all content types).

In summary, this high-level characterization reveals substantial changes in content delivery systems usage in the Internet, as seen from the vantage point of UW. First, the balance of HTTP traffic has changed dramatically over the last several years, with P2P traffic overtaking WWW traffic as the largest contributor to HTTP bytes transferred. Second, although UW is a large publisher of web documents,

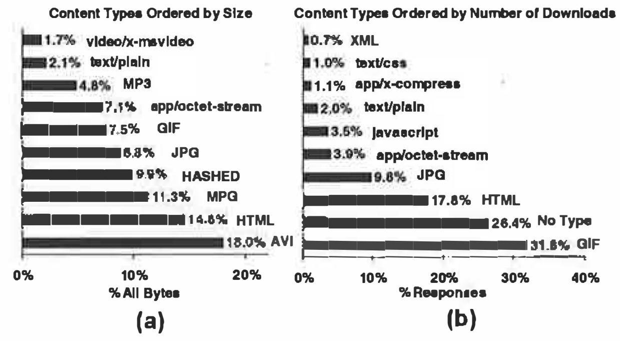


Figure 3. **Content types downloaded by UW clients:** a histogram of the top 10 content types downloaded by UW clients, across all four systems, ordered by (a) size and (b) number of downloads.

P2P traffic makes the University an even larger exporter of data. Finally, the mixture of object types downloaded by UW clients has changed, with video and audio accounting for a substantially larger fraction of traffic than three years ago, despite the small number of requests involving those data types.

5 Detailed Content Delivery Characteristics

The changes in Internet workload that we have observed raise several questions, including: (1) what are the properties of the new objects being delivered, (2) how are clients using the new content-delivery mechanisms, and (3) how do servers for new delivery services differ from those for the web? We attempt to answer these questions in the subsections below.

5.1 Objects

Data in Section 4 suggests that there is a substantial difference in typical object size between P2P and WWW traffic. Figure 4 illustrates this in dramatic detail. Not surprisingly, Akamai and WWW object sizes track each other fairly closely. The median WWW object is approximately 2KB, which matches previous measurement studies [15]. The Kazaa and Gnutella curves are strikingly different from the WWW; the median object size for these P2P systems is

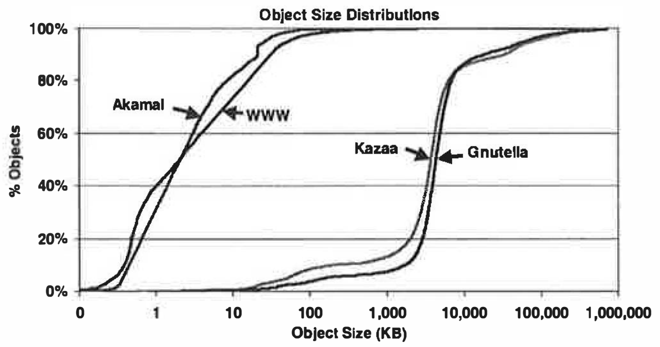


Figure 4. **Object size distributions:** cumulative distributions (CDFs) of object sizes.

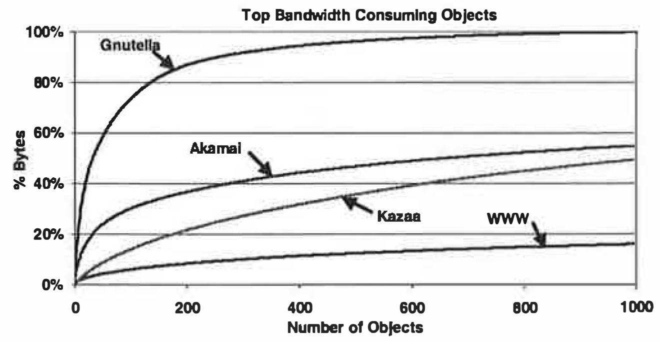


Figure 5. **Top bandwidth consuming objects:** a CDF of bytes fetched by UW clients for top 1,000 bandwidth-consuming objects.

approximately 4MB – a *thousand-fold* increase over the average web document size! Worse, we see that 5% of Kazaa objects are over 100MB. This difference has the potential for enormous impact on Internet performance as these systems grow.

Figure 5 shows a cumulative distribution of bytes fetched by UW clients for the 1,000 highest bandwidth-consuming objects in each of the four CDNs. The Akamai curve rises steeply, with the top 34 objects accounting for 20% of the Akamai bytes transferred; Akamai traffic is clearly skewed to its most popular documents. For Kazaa, we see that a relatively small number of objects account for a large portion of the transferred bytes as well. The top 1,000 Kazaa objects (out of 111K objects accessed) are responsible for 50% of the bytes transferred. For the web, however, the curve is much flatter: the top 1,000 objects only account for 16% of bytes transferred.

To understand this better, we examined the 10 highest bandwidth-consuming objects for WWW, Akamai and Kazaa, which are responsible for 1.9%, 25% and 4.9% of the traffic for each system, respectively. The details are shown in Table 2. For WWW, we see that the top 10 objects are a mix of extremely popular small objects (e.g., objects 1, 2 and 4), and relatively unpopular large objects (e.g., object 3). The worst offender, object 1, is a small object accessed many times. For Akamai, although 8 out of the top

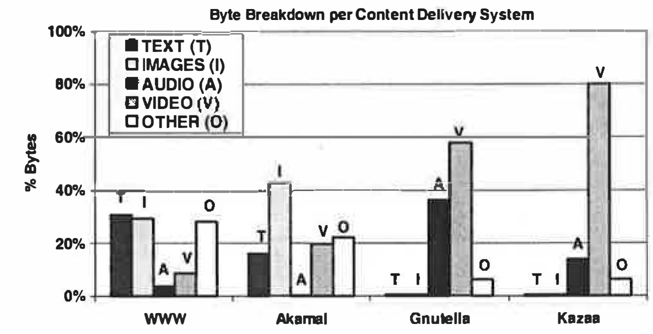


Figure 6. **Downloaded bytes by object type:** the number of bytes downloaded from each system, broken into content type.

10 objects are large and unpopular, 2 out of the top 3 worst offenders are small and popular. Kazaa's inbound traffic, on the other hand, is completely consistent; all of its worst offenders are extremely large objects (on the order of 700MB) that are accessed only ten to twenty times.

Comparing Kazaa inbound and outbound traffic in Table 2 shows several differences. The objects that contribute most to bandwidth consumption in either direction are similarly sized, but UW tends to export these large objects more than it imports them. A small number of UW clients access large objects from a small number of external servers, but nearly thirty times as many external clients access similarly-sized objects from a handful of UW servers, leading to approximately ten times as much bandwidth consumption. This suggests that a *reverse* cache that absorbs outbound traffic might benefit the University even more than a forward cache that absorbs inbound traffic.

Figure 3 in the previous section showed a breakdown of all HTTP traffic by content type for UW-client-initiated traffic. Figure 6 shows a similar breakdown, by bytes, but for each individual CDN. Not surprisingly, the highest component of WWW traffic is text, followed closely by images, while Akamai is dominated by images (42% of bytes are GIF and JPEG). In contrast, Kazaa is completely dominated by video (80%), followed by 14% audio; Gnutella is more evenly split with 58% video and 36% audio.

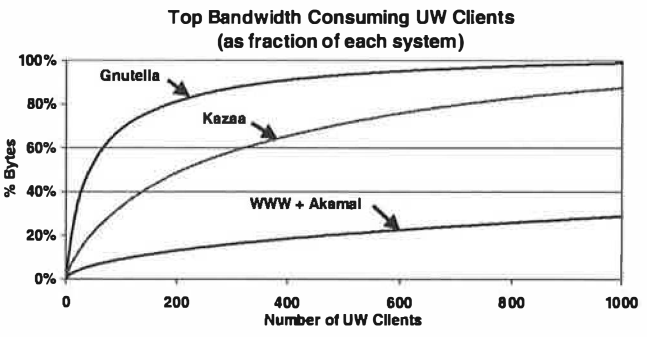
5.2 Clients

The previous subsection considered the characteristics of *what* is transferred (the object view); here we consider *who* is responsible (the client view). Because WWW and Akamai are indistinguishable from a UW client's perspective, this section presents these two workloads combined.

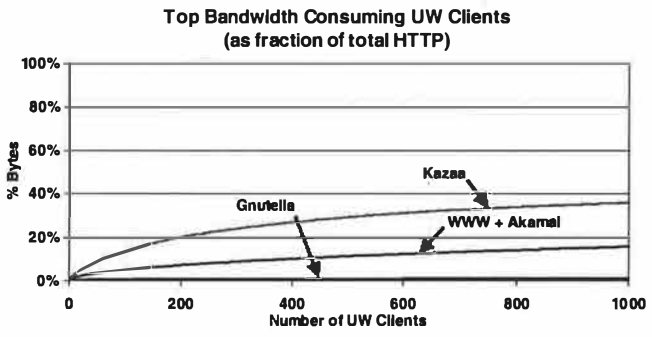
Figure 7a shows a cumulative distribution of bytes downloaded by the top 1000 bandwidth-consuming UW clients for each CDN. It's not surprising that the WWW+Akamai curve is lower; the graph shows only a small fraction of the 39K WWW+Akamai clients, but nearly a quarter of the 4644 Kazaa clients. Nevertheless, in both cases, a small number of clients account for a large portion of the traffic. In the case of the WWW, the top 200 clients (0.5% of the

	WWW (inbound)			Akamai			Kazaa (inbound)				Kazaa (outbound)			
	object size (MB)	GB consumed	# requests	object size (MB)	GB consumed	# requests	object size (MB)	GB consumed	# clients	# servers	object size (MB)	GB consumed	# clients	# servers
1	0.009	12.29	1,412,104	22.37	4.72	218	694.39	8.14	20	164	696.92	119.01	397	1
2	0.002	6.88	3,007,720	0.07	2.37	45,399	702.17	6.44	14	91	699.28	110.56	1000	4
3	333	6.83	21	0.11	1.64	68,202	690.34	6.13	22	83	699.09	78.76	390	10
4	0.005	6.82	1,412,105	9.16	1.59	2,222	775.66	5.67	16	105	700.86	73.30	558	2
5	2.23	3.17	1,457	13.78	1.31	107	698.13	4.70	14	74	634.25	64.99	540	1
6	0.02	2.69	126,625	82.03	1.14	23	712.97	4.69	17	120	690.34	64.97	533	10
7	0.02	2.69	122,453	21.05	1.01	50	715.61	4.49	13	71	690.34	54.90	447	16
8	0.03	1.92	56,842	16.75	1.00	324	579.13	4.30	14	158	699.75	49.47	171	2
9	0.01	1.91	143,780	15.84	0.95	68	617.99	4.12	12	94	696.42	43.35	384	14
10	0.04	1.86	47,676	15.12	0.80	57	167.18	3.83	39	247	662.69	42.28	151	2

Table 2. Top 10 bandwidth consuming objects: the size, bytes consumed, and number of requests (including the partial and unsuccessful ones) for the top 10 bandwidth consuming objects in each system. For Kazaa, instead of requests, we show the number of clients and servers that participated in (possibly partial) transfers of the object.



(a)



(b)

Figure 7. Top UW bandwidth consuming clients: a CDF of bytes downloaded by the top 1000 bandwidth-consuming UW clients (a) as a fraction of each system, and (b) as a fraction of the total HTTP traffic.

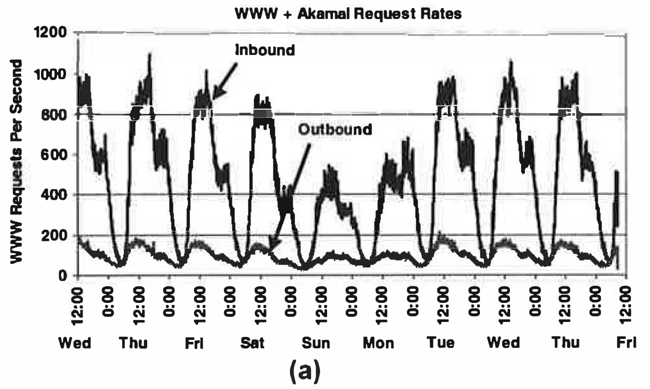
population) account for 13% of WWW traffic; for Kazaa, the top 200 clients (4% of the population) account for 50% of Kazaa traffic. The next 200 Kazaa clients account for another 16% of its traffic. Clearly, a very small number of Kazaa clients have a huge overall bandwidth impact.

To see the impact more globally, the curves in Figure 7b show the *fraction* of the total HTTP bytes downloaded by the most bandwidth-consuming clients for each CDN (the curves are cumulative). This allows us to quantify the impact of a particular CDN's clients on total HTTP traffic. Gnutella clients have almost no impact as consumers of HTTP bandwidth. In contrast, the Kazaa users are the worst offenders: the top 200 Kazaa clients are responsible for 20% of the total HTTP bytes downloaded. In comparison, the top 200 WWW+Akamai clients are responsible for only 7% of total HTTP bytes. Further out, the top 400 Kazaa and WWW clients are responsible for 27% and 10% of total HTTP traffic, respectively.

Given the bandwidth consumed by the web and peer-to-peer delivery systems, it is interesting to examine the request rates that are creating that bandwidth. Figures 8a and 8b show the inbound and outbound request rates for WWW+Akamai and Kazaa, respectively; notice the nearly two order-of-magnitude difference in the Y axis scales. For Kazaa, the outbound request rate peaks at 40 requests per second, dominating the inbound request rate of 23 requests per second. In contrast, the WWW+Akamai inbound request rate peaks at 1100 requests per second, dominating the WWW outbound[1] request rate of just under 200 requests per second. At a high level, then, Kazaa has a request rate about two orders of magnitude lower than the web, but median object size about three orders of magnitude higher than the web. The result is that overall, Kazaa consumes more bandwidth. Similarly, WWW's outbound bandwidth exceeds its inbound bandwidth, despite the opposite trend in request rate; this results from the difference in transfer size in the two directions. While inbound web documents are largely HTML or images, outbound is dominated by application/octet-streams (possibly UW-supplied software, binary data, and video streams from its TV station or web-broadcast technical talks).

A perhaps surprising (although now understandable) result of the disparity in WWW+Akamai and Kazaa object sizes and request rates is shown in Figure 9, which graphs the number of *concurrent* HTTP transactions active at a time for the two systems. Despite the order-of-magnitude

[1]No Akamai data is present in the outbound direction.



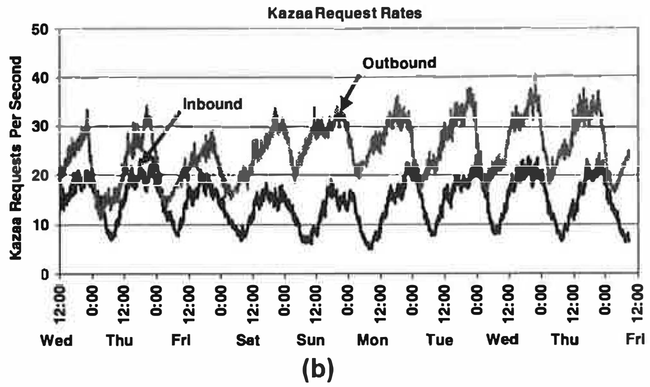


Figure 8. **Request rates over time:** inbound and outbound HTTP transaction rates for (a) the WWW + Akamai, and (b) Kazaa.

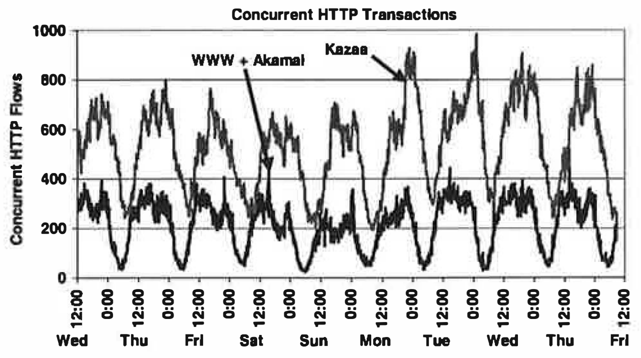


Figure 9. **Concurrent HTTP transactions:** concurrent HTTP transactions for UW Clients.

request-rate advantage of WWW over Kazaa, the number of simultaneous open Kazaa connections is about twice the number of simultaneous open WWW+Akamai connections. While Kazaa generates only 23 requests per second, it is responsible for up to almost 1000 open requests at a time due to its long transfers. Compared to the web requests, whose median duration is 120 ms, the median duration for Kazaa requests is 130 seconds – a 1000-fold increase that tracks the object size. This fact has important implications for the network infrastructure that must maintain those connections.

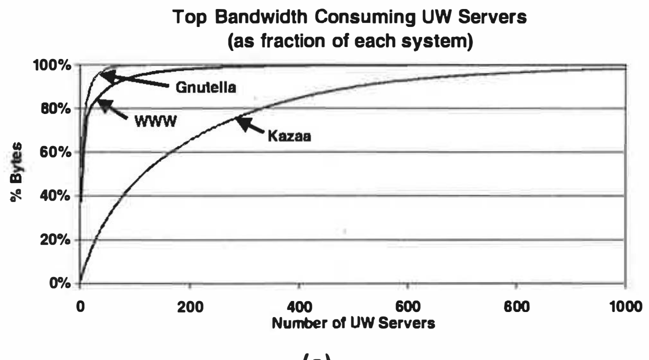
5.3 Servers

This section examines servers: the suppliers of objects and bytes. Figure 10a shows the CDF of bytes transferred by UW-internal servers to external clients. Gnutella has the smallest number of content-serving hosts, and all of the bytes are served by the first 10 of those servers. The WWW curve is quite steep; in this case, the campus has several major servers that provide documents to the Web, and 80% of the WWW traffic is supplied by 20 of the 9821 internal servers we identified. The Kazaa curve rises more slowly, with 80% of the Kazaa bytes being served by the top 334 of the 3888 Kazaa peers we found serving data. One would expect the Kazaa curve to be flatter; an explicit goal of peer-to-peer structures like Kazaa is to spread the load uniformly across the peers. We'll look at this issue in more detail with external Kazaa servers.

Figure 10b shows the fraction of total HTTP bytes transferred (cumulative) from the top UW servers for the CDNs. The global impact of a small number of internal Kazaa peers is clearly seen on the graph. Again, a small number of WWW servers do most of the work for WWW, but this is a small part of the total HTTP outgoing traffic; 20 WWW servers provide 20% of all HTTP bytes output, and the curve rises very slowly from there. However, from the Kazaa curve we see that 170 Kazaa peers are responsible for over 50% of all HTTP bytes transferred; the top 400 Kazaa peers are creating 70% of all outgoing HTTP traffic.

In the opposite direction (UW-external servers), Figures 11a and 11b show the cumulative and HTTP-fractional distributions for incoming bytes transferred, respectively, from the top 1,000 external servers to UW clients. The cumulative distribution (Figure 11a) shows the WWW and Kazaa curves rising very slowly. The WWW curve first rises steeply (for the most popular servers), then levels off, with 938 (out of 400,000) external servers supplying 50% of the total WWW bytes to UW clients; this closely matches our previous findings [35]. The Kazaa curve shows that 600 external Kazaa peers (out of 281,026) supply 26% of the Kazaa bytes to internal peers; this result, however, is somewhat unexpected. Web clients request data from *specific* servers by specifying a URL. A small number of web servers are highly popular, and the popularity curve has a large tail (Zipf) of very unpopular servers. Peer-to-peer systems, though, are different by design. Clients request documents by name, not by server. Those documents may (and hopefully, will) exist on many peers. The goal of the peer-to-peer overlay structure is to broadly distribute work for both scalability and availability. In Kazaa, large files are downloaded by transferring different fragments of the file from different peers, to spread the load among multiple peers. Overall, one would expect the server load for Kazaa to be *much* more widely distributed among peers than for WWW. From Figure 11a, this does not appear to be the case.

As a fraction of total HTTP bytes received by UW



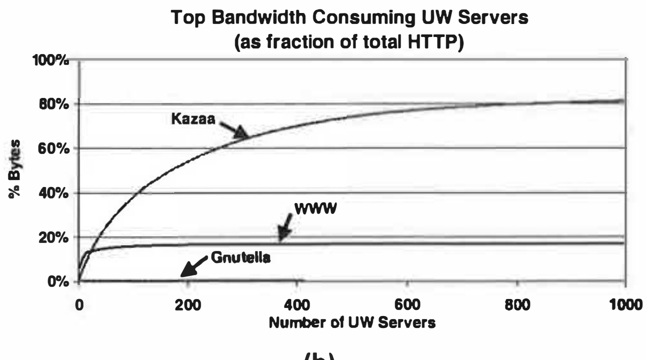
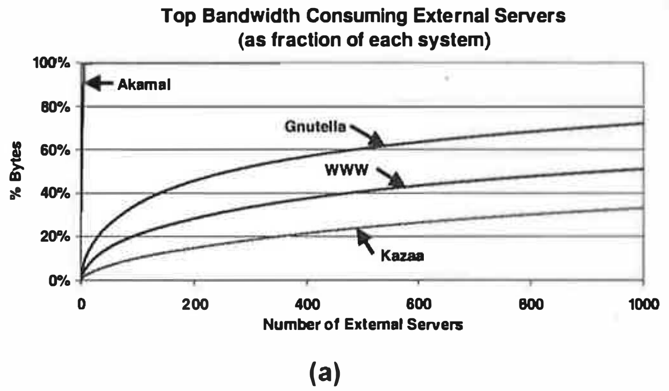


Figure 10. **Top UW-internal bandwidth producing servers:** a CDF of bytes produced by the top 1000 bandwidth-producing UW-internal servers (a) as a fraction of each system, and (b) as a fraction of the total HTTP traffic.



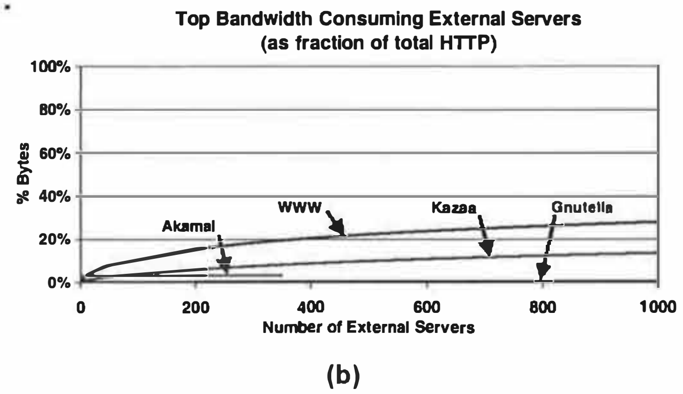


Figure 11. **Top UW-external bandwidth producing servers:** a CDF of bytes produced by the top 1000 bandwidth-producing UW-external servers (a) as a fraction of each system, and (b) as a fraction of the total HTTP traffic.

clients, Figure 11b shows that the top 500 external Kazaa peers supply 10% of the bytes to UW, while the top 500 WWW servers supply 22% of the bytes. Gnutella and Akamai serve an insignificant percentage of the bytes delivered.

Participation in a P2P system is voluntary, and as a result, servers on P2P system are often less well-provisioned than in the web or a CDN [31]. In Figure 12, we show the response codes returned by external servers in each content delivery system. Figure 12a shows that for Akamai and the WWW, approximately 70% of requests result in a successful transaction. However, for P2P systems, less than 20% of requests result in a successful transaction. Most P2P requests are met with a "service unavailable" response, suggesting that P2P servers are often saturated.

Figure 12b shows that nearly *all* HTTP bytes transferred in WWW, Akamai and P2P systems are for useful content. Even though most P2P requests are rejected, the overhead of rejected requests is small compared to the amount of useful data transferred while downloading content.

5.4 Scalability of Peer-to-Peer Systems

The data presented here raises serious questions about whether P2P systems like Kazaa can scale in environments such as the University. Unlike the web, where most clients and servers are separate entities, every peer in a P2P system consumes bandwidth in both directions. Each new P2P client added becomes an immediate server for the entire P2P structure. In addition, we see that the average Kazaa object is huge, and a small number of peers can consume an enormous amount of total network bandwidth in both directions. Over the period of our trace, an average web client consumed 41.9 MB of bandwidth; in contrast, an average Kazaa peer consumed 3.6 GB of bandwidth. Therefore, the bandwidth cost of each Kazaa peer is approximately 90 times that of a web client! This implies that for our environment, adding another 450 Kazaa peers would be equivalent to *doubling* the entire campus web client population. It seems questionable whether any organization providing bandwidth to a large client population can, in the long run, support a service with these characteristics.

5.5 Summary

This section examined our HTTP workload with respect to objects, clients, and servers. From the data presented, we find several important observations:

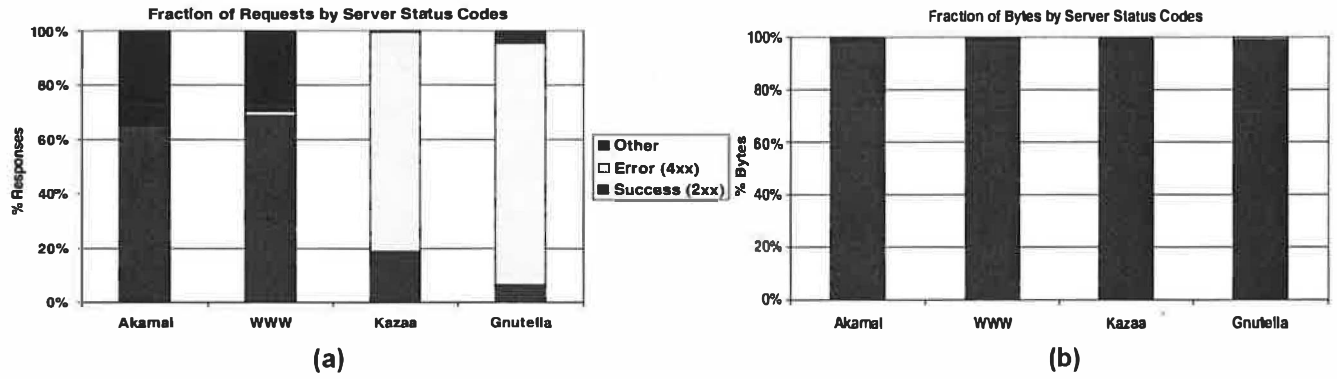


Figure 12. **Server status codes:** response codes returned by external servers; (a) shows the fraction of requests broken down by response code, (b) shows the fraction of bytes broken down by response code.

1. Peer-to-peer traffic, which now accounts for over three quarters of HTTP traffic in our environment, is characterized by objects whose median sizes are three orders of magnitude larger than web objects. It is this object size, rather than widespread usage, that accounts for the large fraction of bytes transferred due to P2P traffic.

2. A small number of P2P users are consuming a disproportionately high fraction of bandwidth. The 200 top Kazaa clients are responsible for 50% of Kazaa bytes downloaded, and nearly 27% of *all* HTTP bytes received by UW.

3. While the P2P request rate is quite low, the transfers last long (three orders of magnitude longer than WWW transfers), resulting in many simultaneous P2P connections. Despite a two order of magnitude difference between P2P and WWW request rates, the number of simultaneous open P2P connections is twice the number of simultaneous open WWW connections.

4. While the design of P2P overlay structures focuses on spreading the workload for scalability, our measurements show that a small number of servers are taking the majority of the burden. In our measurements, only 600 of the 281,026 UW-external Kazaa peers we saw provided 26% of the bytes received by UW clients.

These results have implications for the web as a whole, for the scalability of peer-to-peer systems, and for the potentials for the use of caching in our environment. We focus on this latter issue in the next section.

6 The Potential Role of Caching in CDNs and P2P Systems

Caching in the web is well understood: caches have been shown to absorb bandwidth and reduce access latency [4, 5, 7, 11, 12]. In this section of the paper, we explore caching in the context of the Akamai CDN and Kazaa

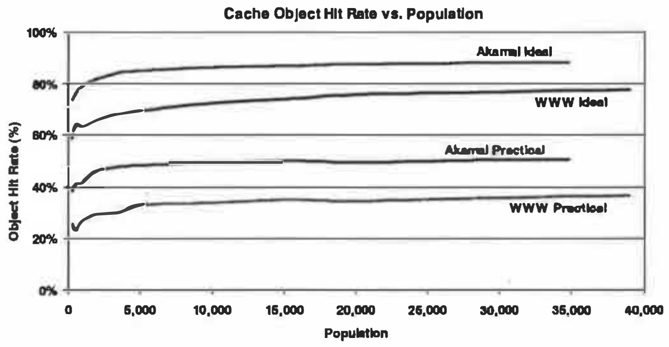


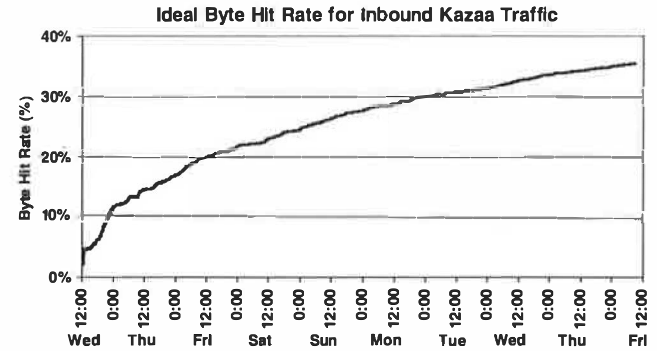
Figure 13. **WWW and Akamai object hit rates vs. population:** object hit rates as a function of population size. An ideal cache treats all documents as cacheable. A practical cache accounts for HTTP 1.1 caching directives. All simulations used an infinite-capacity proxy cache.

P2P systems. For Akamai, we ask whether or not there is a performance advantage of an Akamai CDN relative to a local proxy cache, and for Kazaa, we present an initial study of the effectiveness of caching in that environment.

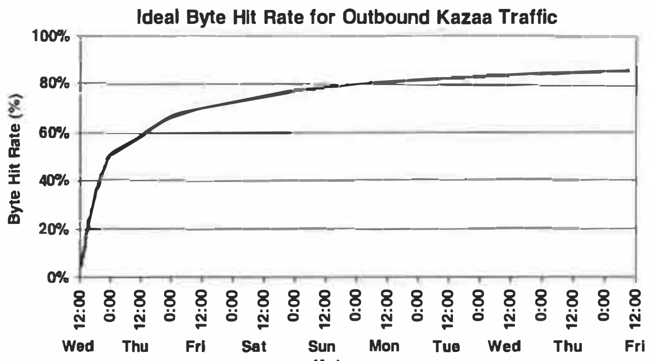
6.1 CDN Caching

Content-delivery networks such as Akamai provide a number of benefits beyond end-user performance, including load balancing, data availability, and reduction of server load. Nonetheless, one interesting question is whether they provide any performance advantage relative to a local proxy cache in an environment such as ours. To answer this question, we simulated the behavior of a UW-local proxy cache against our traced WWW and Akamai workloads.

Figure 13 shows the performance of an infinite-capacity proxy cache for the HTTP transactions in our trace that were sent to Akamai and WWW servers. For both systems, we simulated an "ideal" hit rate (assuming all documents are cacheable) and the hit rate of a "practical" cache. A practical cache accounts for HTTP 1.1 cache control headers [16], cookies, object names with suffixes naming dynamic objects, no-cache pragmas, and uncacheable methods and response codes. For WWW, both the ideal and practical object hit rate curves look similar to the ones from our 1999 study



(a)



(b)

Figure 14. **Ideal Kazaa cache byte hit rate:** cache byte hit rate over time, for (a) inbound traffic (requests from UW clients) and (b) outbound traffic (requests from external clients).

of the same population [35]. In contrast, the Akamai requests achieve an 88% ideal hit rate and a 50% practical hit rate, noticeably higher than WWW requests (77% and 36% respectively). Our analysis shows that Akamai requests are more skewed towards the most popular documents than are WWW requests, i.e., they are less heavy-tailed, with a Zipf parameter of 1.08 vs. 0.7 for the WWW.

The ideal hit rate for Akamai traffic is extremely high. While many Akamai objects are marked as uncacheable (reflected in the low hit rate of a cache respecting caching directives), we know that most bytes fetched from Akamai are from images and videos; this implies that much of Akamai's content is in fact static and could be cached.

Overall, this analysis indicates that a local web proxy cache could achieve nearly the same hit rate as an Akamai replica (which presumably has a hit rate of 100%), considering in particular that the misses in our proxy are primarily cold-cache effects. Therefore, we would expect that widely-deployed proxy caches would significantly reduce the need for a separate content delivery network, at least with respect to local performance, assuming that immutable objects could be marked as cacheable with long time-to-live values (TTLs).

6.2 P2P Caching

Given the P2P traffic volume that we observed, the potential impact of caching in P2P systems may exceed the benefits seen in the web. In this section, we present an initial exploration of P2P caching. Our goal is not to solve (or even identify) all of the complexities of P2P caching – a full caching study would be outside of the scope of this paper – but rather to gain insight into how important a role caching may play.

To explore this question, we built an ideal (i.e., infinite capacity and no expiration) cache simulator for Kazaa P2P traffic. Since the average P2P object is three orders of magnitude larger than the average web object, we evaluate the benefits of a P2P cache with respect to its byte hit rate, rather than its object hit rate. Because Kazaa peers can transfer partial fragments as well as entire objects, our cache stores items at the granularity of 32 KB *blocks*. For each Kazaa transfer, we identified and cached all complete 32 KB (aligned) blocks. Future requests for the same object may be partially satisfied from these cached blocks. Because of this, a single Kazaa transfer may involve multiple cache block hits and misses.

Figures 14a and 14b show the byte hit rate over time for inbound and outbound Kazaa traffic, respectively. The outbound hit rate (Figure 14b) increases as the cache warms, stabilizing at approximately 85%. This is a remarkably high hit rate – double that reported for web traffic. A reverse P2P cache deployed in the University's ISP would result in a peak bandwidth savings of over 120 megabits per second!

The inbound hit rate grows more slowly over time, reaching only 35% by the end of the trace. It is clear that the simulated inbound cache has not fully warmed even for nine days worth of traffic. Accordingly, we will comment only on outbound traffic for the remainder of this section.

We investigated the source of the high outbound hit rate by examining the 300 top bandwidth-consuming objects. These objects had an average size of 614 MB and a total size of 180 GB. Requests for these 300 objects consumed approximately 5.635 TB of traffic throughout the trace, which is 42% of the total bytes consumed by Kazaa outbound traffic. A conservative estimate suggests that a cache serving only these 300 objects would see a byte hit rate of more than 38% when presented with our entire trace. Thus, a small number of large objects are the largest contributors to the outbound P2P cache byte hit rate.

In Figure 15, we show the cache byte hit rate as a function of population size for outbound traffic. A population of 1,000 clients sees a hit rate of 40%; hit rate climbs slowly thereafter, until a population of 500,000 clients sees a hit rate of 85%. This indicates that a P2P cache would be effective for a small population, but even more effective for large populations.

Currently, many organizations are either filtering or rate-limiting P2P traffic to control bandwidth consumption. Based on our preliminary investigation in this section, we believe caching would have a large effect on a wide-scale

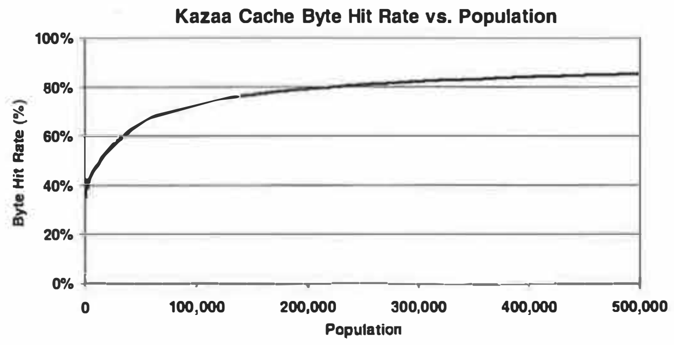


Figure 15. Kazaa cache byte hit rate vs. population: byte hit rate as a function of population size, for outbound traffic.

P2P system, potentially reducing wide-area bandwidth demands dramatically.

7 Conclusions

This paper examined Internet content delivery systems from the perspective of traffic flowing in and out of the University of Washington. To do this, we gathered a trace of all incoming and outgoing traffic over a period of nine days, separating out the HTTP traffic components due to standard web traffic, Akamai-supplied content, Gnutella P2P file transfers, and Kazaa P2P file transfers. Our results confirm that a dramatic shift in Internet traffic and usage has occurred in only several years. Specifically, for our environment we found that:

- Peer-to-peer traffic now accounts for the majority of HTTP bytes transferred, exceeding traffic due to WWW accesses by nearly a factor of three. As a fraction of all TCP traffic, Kazaa alone accounts for 36.9% of bytes transferred, compared to 14.3% for web documents.
- P2P documents are three orders of magnitude larger than web objects, leading to a 1000-fold increase in transfer time. As a result, the number of concurrent P2P flows through the University is approximately twice the number of flows for our web population, despite the extremely low request rate and small population of P2P systems.
- A small number of extremely large objects account for an enormous fraction of observed P2P traffic. For Kazaa transfers out of the University, the top 300 objects, with a total size of 180 GB, were responsible for 5.64 TB of the traffic – almost half of the total outbound Kazaa bandwidth.
- A small number of clients and servers are responsible for the majority of the traffic we saw in the P2P systems. The top 200 of 4,644 UW Kazaa clients accounted for 50% of downloaded Kazaa bytes. More surprisingly, only 600 UW-external peers (out of the 281,026 servers used) provided 26% of the bytes transferred into the University.
- Each P2P client creates a significant bandwidth load in *both* directions, with uploads exceeding downloads for Kazaa users. Our 4,644 Kazaa peers provided 13.573 TB of data to 611,005 external peers, while requesting 1.78 TB of data from 281,026 peers. Overall, the bandwidth requirement of a single Kazaa peer is ninety-fold that of a single web client.

Overall, these points indicate that despite the scalability-based design, the bandwidth demands of peer-to-peer systems such as Kazaa will likely prevent them from scaling further, at least within University-like environments. However, our initial analysis also shows that an organizational P2P proxy cache has the potential to significantly reduce P2P bandwidth requirements. We intend to examine caching in more depth in our future work.

Acknowledgements

We would like to thank Brian Youngstrom who helped us immensely to make our traces a reality. We would also like to thank David Richardson, Art Dong and the other members of the Computing and Communications organization at UW for their support. We are grateful for the guidance of David Culler, our shepherd, and our anonymous reviewers. We thank Alec Wolman and Geoffrey Voelker for their help with the tracing infrastructure. This research was supported in part by NSF grants CCR-0121341, ITR-0085670, and IIS-0205635.

References

[1] Akamai. `http://www.akamai.com`.

[2] J. Almeida, V. Almeida, and D. Yates. Measuring the behavior of a world-wide web server. Technical Report 1996-025, Boston University, Oct. 1996.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proc. of IEEE INFOCOM 1999*, March 1999.

[4] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: The devil is in the details. In *Workshop on Internet Server Performance*, June 1998.

[5] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Proc. of IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing*, Sep. 1998.

[6] S. Chakrabarti, B.E. Dom, S. R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins, D. Gibson, and J. Kleinberg. Mining the Web's link structure. *Computer*, 32(8), 1999.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *Proc. of the 1996 USENIX Annual Technical Conf.*, Jan. 1996.

[8] M. Chesire, A. Wolman, G. Voelker, and H. Levy. Measurement and analysis of a streaming media workload. In *Proc. of the 2001 USENIX Symp. on Internet Technologies and Systems*, March 2001.

[9] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[10] Clip2. The Gnutella protocol specification v.0.4, March 2001. `http://www.clip2.com/GnutellaProtocol04.pdf`.

[11] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. C. Mogul. Rate of change and other metrics: a live study of the world wide web. In *Proc. of the 1997 USENIX Symp. on Internet Technologies and Systems*, Dec. 1997.

[12] B. Duska, D. Marwood, and M. J. Feeley. The measured access characteristics of World Wide Web client proxy caches. In *Proc. of the 1st USENIX Symp. on Internet Technologies and Systems*, Dec. 1997.

[13] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proc. of ACM SIGCOMM '98*, Aug. 1998.

[14] S. Gadde, J. Chase, and M. Rabinovich. Web caching and content distribution: A view from the interior. In *Proc. of the 5th International Web Caching and Content Delivery Workshop*, May 2000.

[15] S. D. Gribble and E. A. Brewer. System design issues for internet middleware services: Deductions from a large client trace. In *Proc. of the 1997 USENIX Symp. on Internet Technologies and Systems*, Dec. 1997.

[16] Internet Engineering Task Force. Hypertext transfer protocol - http 1.1. RFC 2068, March 1997.

[17] Internet2. `http://netflow.internet2.edu/weekly/20020422`.

[18] K. L. Johnson, J. F. Carr, M. S. Day, and M. Frans Kaashoek. The measured performance of content distribution networks. *Computer Communications*, 24(2), 2001.

[19] J. Kangasharju, K.W. Ross, and J.W. Roberts. Performance evaluation of redirection schemes in content distribution networks. *Computer Communications*, 24(2):207–214, 2001.

[20] Kazaa. `http://www.kazaa.com`.

[21] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins. The Web as a graph: Measurements, models, and methods. In T. Asano, H. Imai, D. T. Lee, S. Nakano, and T. Tokuyama, editors, *Proc. of the 5th Annual Int. Conf. Computing and Combinatorics*, number 1627. Springer-Verlag, 1999.

[22] M. Koletsou and G. M. Voelker. The Medusa proxy: A tool for exploring user-perceived web performance. In *Proc. of the Sixth Int. Workshop on Web Caching and Content Distribution*, June 2001.

[23] B. Krishnamurthy, C. Wills, and Y. Zhang. On the use and performance of content distribution networks. In *Proc. of SIGCOMM IMW 2001*, Nov. 2001.

[24] N. Leibowitz, A. Bergman, R. Ben-Shaul, and A. Shavit. Are file swapping networks cacheable? Characterizing P2P traffic. In *Proc. of the 7th Int. WWW Caching Workshop*, August 2002.

[25] B. Maggs. Global Internet Content Delivery. Talk delivered in the Internet and Distributed Systems Seminar at Stanford University. `http://www.stanford.edu/class/cs548/abstracts.shtml#bruce`.

[26] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of the USENIX Technical Conf.*, Winter 1993.

[27] J.-M. Menaud, V. Issarny, and M. Banatre. A new protocol for efficient transversal Web caching. In *Proc. of the 12th Int. Symp. on Distributed Computing*, Sep. 1998.

[28] Napster. `http://www.napster.com`.

[29] V. N. Padmanabhan and L. Qiu. The content and access dynamics of a busy web site: Findings and implications. In *Proc. of ACM SIGCOMM 2000*, August 2000.

[30] M. Rabinovich, J. Chase, and S. Gadde. Not all hits are created equal: Cooperative proxy caching over a wide area network. In *Proc. of the 3rd Int. WWW Caching Workshop*, June 1998.

[31] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking 2002*, Jan. 2002.

[32] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of DNS-based server selection. In *Proc. of IEEE INFOCOM 2001*, Anchorage, AK, USA 2001.

[33] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design considerations for distributed caching on the Internet. In *The 19th IEEE Int. Conf. on Distributed Computing Systems*, May 1999.

[34] D. Wessels, K. Claffy, and H.-W. Braun. NLANR prototype web caching system. `http://ircache.nlanr.net/`.

[35] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of web-object sharing and caching. In *Proc. of the 2nd USENIX Conf. on Internet Technologies and Systems*, Oct. 1999.

[36] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. The scale and performance of cooperative web proxy caching. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, Dec. 1999.

[37] B. Yang and H. Garcia-Molina. Efficient search in peer-to-peer networks. In *Proc. of the 22nd Int. Conf. on Distributed Computing Systems*, July 2002.

TCP Nice: A Mechanism for Background Transfers

Arun Venkataramani Ravi Kokku Mike Dahlin *

Laboratory of Advanced Systems Research
Department of Computer Sciences
University of Texas at Austin, Austin, TX 78712
{arun, rkoku, dahlin}@cs.utexas.edu

Abstract

Many distributed applications can make use of large *background transfers* – transfers of data that humans are not waiting for – to improve availability, reliability, latency or consistency. However, given the rapid fluctuations of available network bandwidth and changing resource costs due to technology trends, hand tuning the aggressiveness of background transfers risks (1) complicating applications, (2) being too aggressive and interfering with other applications, and (3) being too timid and not gaining the benefits of background transfers. Our goal is for the operating system to manage network resources in order to provide a simple abstraction of near zero-cost background transfers. Our system, TCP Nice, can provably bound the interference inflicted by background flows on foreground flows in a restricted network model. And our microbenchmarks and case study applications suggest that in practice it interferes little with foreground flows, reaps a large fraction of spare network bandwidth, and simplifies application construction and deployment. For example, in our prefetching case study application, aggressive prefetching improves demand performance by a factor of three when Nice manages resources; but the same prefetching hurts demand performance by a factor of six under standard network congestion control.

1 Introduction

Many distributed applications can make use of large *background transfers* – transfers of data that humans are not waiting for – to improve service quality. For example, a broad range of applications and services such as data backup [29], prefetching [50], enterprise data distribution [20], Internet content distribution [2], and peer-to-peer storage [16, 43] can trade increased network bandwidth consumption and possibly disk space for improved service latency [15, 18, 26, 32, 38, 50], improved availability [11, 53], increased scalability [2], stronger consistency [53], or support for mobility [28, 41, 47]. Many of these services have potentially unlimited bandwidth demands where incrementally more bandwidth consumption provides incrementally better service. For example, a web prefetching system can improve its hit rate by fetching objects from a virtually unlimited collection of objects that have non-zero probability of access [8, 10] or by updating cached copies more frequently as data change [13, 50, 48]; Technology trends suggest that "wasting" bandwidth and storage to improve latency and availability will become increasingly attractive in the future: per-byte network transport costs and disk storage costs are low and have been improving at 80-100% per year [9, 17, 37]; conversely network availability [11, 40, 54] and network latencies improve slowly, and long latencies and failures waste human time.

Current operating systems and networks do not provide good support for aggressive background transfers. In particular, because background transfers compete with foreground requests, they can hurt overall performance and availability by increasing network congestion. Applications must therefore carefully balance the benefits of background transfers against the risk of both *self-interference*, where applications hurt their own performance, and *cross-interference*, where applications hurt other applications' performance. Often, applications attempt to achieve this balance by setting "magic numbers" (e.g., the prefetch threshold in prefetching algorithms [18, 26]) that have little obvious relationship to system goals (e.g., availability or latency) or constraints (e.g., current spare network bandwidth).

Our goal is for the operating system to manage network resources in order to provide a simple abstraction of zero-cost background transfers. A self-tuning background transport layer will enable new classes of applications by (1) simplifying applications, (2) reducing the risk of being too aggressive, and (3) making

*This work was supported in part by an NSF CISE grant (CDA-9624082), the Texas Advanced Technology Program, the Texas Advanced Research Program, and Tivoli. Dahlin was also supported by an NSF CAREER award (CCR-9733842) and an Alfred P. Sloan Research Fellowship.

it easier to reap a large fraction of spare bandwidth to gain the advantages of background transfers. Self-tuning resource management seems essential for coping with network conditions that change significantly over periods of seconds (e.g., changing congestion [54]), hours (e.g., diurnal patterns), and months (e.g., technology trends [9, 37]). We focus on managing network resources rather than processors, disks, and memory both because other work has provided suitable end-station schedulers for these local resources [10, 24, 33, 39, 45] and because networks are shared across applications, users, and organizations and therefore pose the most critical resource management challenge to aggressive background transfers.

Our system, TCP Nice, dramatically reduces the interference inflicted by background flows on foreground flows. It does so by modifying TCP congestion control to be more sensitive to congestion than traditional protocols such as TCP Reno [30] or TCP Vegas [7] by detecting congestion earlier, reacting to it more aggressively, and allowing much smaller effective minimum congestion windows. Although each of these changes is simple, the combination is carefully constructed to provably bound the interference of background flows on foreground flows while still achieving reasonable throughput in practice. Our Linux implementation of Nice allows senders to select Nice or standard Reno congestion control on a connection-by-connection basis, and it requires no modifications at the receiver.

Our goals are to minimize damage to foreground flows while reaping a significant fraction of available spare network capacity. We evaluate Nice against these goals using theory, microbenchmarks, and application case studies.

Because our first goal is to avoid interference regardless of network conditions or application aggressiveness, our protocol must rest on a sound theoretical basis. In Section 3, we argue that our protocol is always less aggressive than Reno, and we prove under a simplified network model that Nice flows interfere with Reno flows' bandwidth by a factor that falls exponentially with the size of the buffer at the bottleneck router independent of the number of Nice flows in the network. Our analysis shows that all three features described above are essential for bounding interference.

Our microbenchmarks comprise both *ns* [36] simulations to stress test the protocol and Internet measurements to examine the system's behavior under realistic conditions. Our simulation results in Section 4 indicate that Nice avoids interfering with Reno or Vegas flows across a wide range of background transfer loads and spare network capacity situations. For example, in one microbenchmark, 16 Nice background flows slow down the average demand document transfer time by less than 10% and reap over 70% of the spare network bandwidth. But in the same situation, 16 backlogged Reno (or Vegas) flows slow demand requests by more than an order of magnitude.

Our Internet microbenchmarks in Section 5 measure the performance of simultaneous foreground and background transfers across a variety of Internet links. We find that background flows cause little interference to foreground traffic: the foreground flows' average latency and bandwidth are little changed between when foreground flows compete with background flows and when they do not. Furthermore, we find that there is sufficient spare capacity that background flows reap significant amounts of bandwidth throughout the day. For example, during most hours Nice flows between London England and Austin Texas averaged more than 80% of the bandwidth achieved by Reno flows; during the worst hour observed they still saw more than 30% of the Reno flows' bandwidth.

Finally, our case study applications seek to examine the end-to-end effectiveness, the simplicity, and the usefulness of Nice. We examine two services. First, we implement a HTTP prefetching client and server and use Nice to regulate the aggressiveness of prefetching. Second, we study a simplified version of the Tivoli Data Exchange [20] system for replicating data across large numbers of hosts. In both cases, Nice allows us to (1) simplify the application by eliminating magic numbers, (2) reduce the risk of interfering with demand transfers, and (3) improve the effectiveness of background transfers by using significant amounts of bandwidth when spare capacity exists. For example, in our prefetching case study, when applications prefetch aggressively, they can improve their performance by a factor of 3 when they use Nice, but if they prefetch using TCP-Reno instead, they overwhelm the network and increase total demand response times by more than a factor of six.

The primary limitation of our analysis is that we evaluate our system when competing against Reno and Vegas TCP flows, but we do not systematically evaluate it against other congestion control protocols such as equation-based [22] or rate-based [42]. Our protocol is strictly less aggressive than Reno, and we expect that it causes little interference with other demand flows, but future work is needed to provide evidence to support this assertion. A second concern is incentive compatibility: will users use low priority flows for background traffic when they could use high priority flows instead? We observe that most of the "aggressive replication" applications cited above do, in fact, voluntarily limit their aggressiveness by, for example, prefetching only objects whose priority of use exceeds a threshold [18, 50]. Two factors may account for this phenomenon. First, good engineers may consider the social costs of background transfers and therefore be conservative in their demands. Second, most users have an incentive to at least avoid self-interference where a user's background traffic interferes with that user's foreground traffic from the same or different application. We thus believe that Nice is a useful tool for both responsible and selfish engineers and users.

The rest of this paper proceeds as follows. Section 2 describes the Nice congestion control algorithm. Sections 3, 4, and 5 describe our analytic results, NS microbenchmark results, and Internet measurement results respectively. Section 6 describes our experience with case study applications. Finally, Section 7 puts this work in context with related work, and Section 8 presents our conclusions.

2 Design and Implementation

In designing our system, we seek to balance two conflicting goals. An ideal system would (1) cause no interference to demand transfers and (2) consume 100% of available spare bandwidth. In order to provide a simple and safe abstraction to applications, we emphasize the former goal and will be satisfied if our protocol makes use of a significant fraction of spare bandwidth. Although it is easy for an adversary to construct scenarios where Nice does not get any throughput in spite of there being sufficient spare capacity in the network, our experiments confirm that in practice, Nice obtains a significant fraction of the throughput of Reno or Vegas when there is spare capacity in the network.

2.1 Background: Existing Algorithms

Congestion control mechanisms in existing transmission protocols are composed of a *congestion signal* and a *reaction policy*. The congestion control algorithms in popular variants of TCP (Reno, NewReno, Tahoe, SACK) use packet loss as a congestion signal. In steady state, the reaction policy uses additive increase and multiplicative decrease (AIMD) in which the sending rate is controlled by a congestion window that is multiplicatively decreased by a factor of two upon a packet drop and is increased by one per window of data acknowledged. The AIMD framework is believed to be fundamental to the robustness of the Internet [12, 30].

However, with respect to our goal of minimizing interference, this congestion signal – a packet loss – arrives too late to avoid damaging other flows. In particular, overflowing a buffer (or filling a RED router enough to cause it to start dropping packets) may trigger losses in other flows, forcing them to back off multiplicatively and lose throughput.

In order to detect incipient congestion due to interference we monitor round-trip delays of packets and use increasing round-trip delays as a signal of congestion. In this respect, we draw inspiration from TCP Vegas [7], a protocol that differs from TCP-Reno in its congestion avoidance phase. By monitoring round-trip delays, each Vegas flow tries to keep between α (typically 1) and β (typically 3) packets buffered at the bottleneck router. If fewer than α packets are queued, Vegas increases the window by one per window of data acknowledged. If more than β packets are queued, the algorithm decreases the window by one per window of data acknowledged. Vegas adjusts the window W once every round as follows ($minRTT$ is the minimum value of all measured round-trip delays and $observedRTT$ is the round-trip delay experienced by a distinguished packet in the previous round):

```
$E \leftarrow \frac{W}{minRTT}$            // Expected throughput
$A \leftarrow \frac{W}{observedRTT}$       // Actual throughput
$Diff \leftarrow (E - A) \cdot minRTT$
if ($Diff < \alpha$)
        $W \leftarrow W + 1$
else if ($Diff > \beta$)
        $W \leftarrow W - 1$
```

Bounding the difference between the actual and expected throughput translates to maintaining between α and β packets in the bottleneck router. Although Vegas seems a promising candidate protocol for background flows, it has some drawbacks: (i) Vegas has been designed to compete for throughput approximately fairly with Reno, (ii) Vegas backs off when the number of queued packets from its flow increases, but it does not necessarily back off when the number of packets enqueued by other flows increase, (iii) each Vegas flow tries to keep 1 to 3 packets in the bottleneck queue, hence a collection of background flows could cause significant interference.

Note that even setting α and β to very small values does not prevent Vegas from interfering with cross traffic. The linear decrease on the "$Diff > \beta$" trigger is not responsive enough to keep from interfering with other flows. We confirm this intuition using simulations and Internet experiments, and it also follows as a conclusion from the theoretical analysis.

2.2 TCP Nice

The Nice extension adds three components to Vegas: first, a more sensitive congestion detector; second, multiplicative reduction in response to increasing round trip times; and third, the ability to reduce the congestion window below one. These additions are simple, but our analysis and experiments demonstrate that the omission of any of them would fundamentally increase the interference caused by background flows.

A Nice flow monitors round-trip delays, estimates the total queue size at the bottleneck router, and signals congestion when this total queue size exceeds a fraction of the estimated maximum queue capacity. Nice uses $minRTT$, the minimum observed round trip time, as the estimate of the round trip delay when queues are empty, and it uses $maxRTT$ as an estimate of the round trip time when the bottleneck queue is full. If more than *fraction* of the packets Nice sends during a RTT window encounter delays exceeding $minRTT + (maxRTT - minRTT) \cdot threshold$, our detector signals congestion. Round-trip delays of packets are indicative of the current bottleneck queue size and the threshold represents the fraction of the total queue capacity that starts to trigger congestion. The Nice congestion avoidance mechanism incorporating the *interference trigger* with threshold t and fraction f can be written as follows ($curRTT$ is the round-trip delay experienced by each packet):

```
per ack operation:
    if ($curRTT > (1-t) \cdot minRTT + t \cdot maxRTT$)
        $numCong$++;
per round operation:
    if ($numCong > f \cdot W$)
        $W \leftarrow W/2$
    else {
        ...    // Vegas congestion avoidance follows
    }
```

If the congestion condition does not trigger, Nice falls back on Vegas' congestion avoidance rules. If a packet is lost, Nice falls back on Reno's rules. The final change to congestion control is to allow the window sizes to multiplicatively decrease below one, if so dictated by the congestion trigger and response. In order to affect window sizes less than one, we send a packet out after waiting for the appropriate number of smoothed round-trip delays.

Maintaining a window of less than one causes us to lose *ack-clocking*, but the flow continues to send at most as many packets into the network as it gets out. In this phase the packets act as network probes waiting for congestion to dissipate. By allowing the window to go below one, Nice retains the non-interference property even for a large number of flows. Both our analysis and our experiments confirm the importance of this feature: this optimization significantly reduces interference, particularly when testing against several background flows. A similar optimization has been suggested even for regular flows to handle cases when the number of flows starts to approach the bottleneck router buffer size [35].

When a Nice flow signals congestion, it halves its current congestion window. In contrast Vegas reduces its window by one packet each round that encounters long round trip times and only halves its window if packets are lost (falling back on Reno-like behavior.) The combination of more aggressive detection and more aggressive reaction may make it more difficult for Nice to maximize utilization of spare capacity, but our design goals lead us to minimize interference even at the potential cost of utilization. Our experimental results show that even with these aggressively timid policies, we achieve reasonable levels of utilization in practice.

As in TCP Vegas, maintaining running measures of $minRTT$ and $maxRTT$ have their limitations - for example, if the network is in a state of persistent congestion, a bad estimate of $minRTT$ is likely to be obtained. However, past studies [1, 44] have indicated that a good estimate of the minimum round-trip delay can typically be obtained in a short time; our experience supports this claim. The use of minimum and maximum values makes the prototype sensitive to outliers. Using the first and ninety-ninth percentile values could improve the robustness of this algorithm, but we have not tested this optimization. Route changes during a transfer can also contribute to inaccuracies in RTT estimates. However such changes are uncommon [40] and we speculate that they can be handled by maintaining exponentially decaying averages for $minRTT$ and $maxRTT$ estimates.

2.3 Prototype Implementation

We implement a prototype Nice system by extending an existing version of the Linux kernel that supports Vegas congestion avoidance. Like Vegas, we use microsecond resolution timers to monitor round-trip delays of packets to implement a congestion detector. In our implementation of Nice, we set the corresponding Vegas parameters α and β to 1 and 3 respectively. After the first round-trip delay estimate, maxRTT is initialized to $2 \cdot minRTT$.

The Linux TCP implementation maintains a minimum window size of two in order to avoid delayed acknowledgements by receivers that attempt to send one acknowledgement every two packets. In order to allow the congestion window to go to one or below one, we add a new timer that runs on a per-socket basis when the congestion window for the particular socket is below two. When in this phase, the flow waits for the appropriate number of RTTs before sending two packets into the network. Thus, a window of 1/16 means that the flow sends out two packets after waiting for 32 smoothed round-trip times. We limit the minimum window size to $1/48$ in our prototype.

Our congestion detector signals congestion when more than *fraction* $= 0.5$ packets during an RTT encounter delays exceeding $threshold = 0.2$. We discuss the sensitivity to $threshold$ in more detail in Section 3. The *fraction* does not enter directly into our analysis; our experimental studies in Section 4 indicate that the interference is relatively insensitive to the *fraction* parameter chosen. Since packets are sent in bursts, most packets in a round observe similar round-trip times. In the future we plan to study pacing packets across a round in order to obtain better samples of prevailing round-trip delays.

Our prototype provides a simple API to designate a flow as a background flow through an option in the *setsockopt* system call. By default, flows are foreground flows.

3 Analysis

Experimental evidence alone is insufficient to allow us to make strong statements about Nice's non-interference properties for general network topologies, background flow workloads, and foreground flow workloads. We therefore analyze it formally to bound the reduction in throughput that Nice imposes on foreground flows. Our primary result is that under a simplified network model, for long transfers, the reduction in the throughput of Reno flows is asymptotically bounded by a factor that falls exponentially with the maximum queue length of the bottleneck router irrespective of the number of Nice flows present.

Theoretical analysis of network protocols, of course, has limits. In general, as one abstracts away details to gain tractability or generality, one risks omitting important behaviors. Most significantly, our formal analysis assumes a simplified fluid approximation and synchronous network model, as described below. Also, our formal analysis holds for long background flows, which are the target workload of our abstraction. But it also assumes long foreground Reno flows, which are clearly not the only cross-traffic of interest. Finally, in our analysis, we abstract detection by assuming that at the end of each RTT epoch, a Nice sender accurately estimates the queue length during the previous epoch. Although these assumptions are restrictive, the insights gained in the analysis lead us to expect the protocol to work well under more general circumstances. The analysis has also guided our design, allowing us to include features that are necessary for noninterference while excluding those that are not. Our experience with the prototype has supported the benefit of using theoretical analysis to guide our design: we encountered few surprises and required

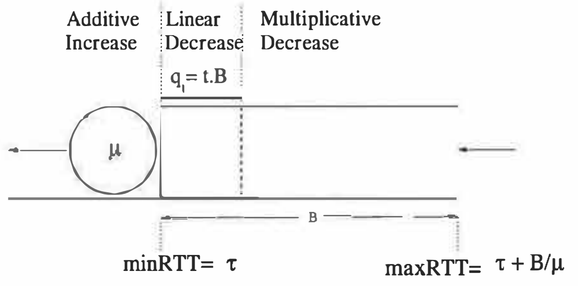


Figure 1: Nice Queue Dynamics

no topology or workload-dependent tuning during our experimental effort.

We use a simplified fluid approximation model of the network to help us model the interaction of multiple flows using separate congestion control algorithms. This model assumes infinitely small packets. We simplify the network itself to a source, destination, and a *single bottleneck*, namely a router that performs drop-tail queuing as shown in Figure 1. Let μ denote the service rate of the queue and B the buffer capacity at the queue. Let τ be the round-trip delay of packets between the source and destination excluding all queuing delays. We consider a fixed number of connections, m following Reno and l following Nice, each of which has one continuously backlogged flow between a source and a destination. Let t be the Nice threshold and $q_t = t \cdot B$ be the corresponding queue size that triggers multiplicative backoff for Nice flows. The connections are homogeneous, *i.e.* they experience the same propagation delay τ. Moreover, the connections are synchronized so that in the case of buffer overflow, all connections simultaneously detect a loss and multiply their window sizes by γ. Models assuming flow synchronization have been used in previous analyses [6]. We model only the congestion avoidance phase to analyze the steady-state behavior.

We obtain a bound on the reduction in the throughput of Reno flows due to the presence of Nice flows by analyzing the dynamics of the bottleneck queue. We achieve this goal by dividing the duration of the flows into *periods*. In each period we bound the decrease in the number of Reno packets processed by the router due to interfering Nice packets. In the following we give an outline of this analysis. The complete analysis with detailed proofs appears in the our technical report [49].

Let $W_r(t)$ and $W_n(t)$ denote respectively the total number of outstanding Reno and Nice packets in the network at time t. $W(t)$, the total window size, is $W_r(t)+W_n(t)$. We trace these window sizes across periods. *The end of a period and the beginning of the next is marked by a packet loss*, at which time each flow reduces its window size by a factor of γ. $W(t) = \mu\tau + B$ just before a loss and $W(t) = (\mu\tau + B) \cdot \gamma$ just after. Let t_0 be the beginning of one such period after a loss. Consider the case when $W(t_0) = (\mu\tau + B)\gamma < \mu\tau$ and $m > l$. For ease of analysis we assume that the "Vegas β" parameter for the Nice flows is 0, *i.e.* the Nice flows additively decrease upon observing round-trip times greater than τ. The window dynamics in any period can be split into three intervals as described below.

Additive Increase, Additive Increase: In this interval $[t_0, t_1]$ both Reno and Nice flows increase linearly. $W(t)$ increases from $W(t_0)$ to $W(t_1) = \mu\tau$, at which point the queue starts building.

Additive Increase, Additive Decrease: This interval $[t_1, t_2]$ is marked by additive increase of W_r, but additive decrease of W_n as the "*Diff* $> \beta$" rule triggers the underlying Vegas controls for the Nice flows. The end of this interval is marked by $W(t_2) = \mu\tau + q_t$.

Additive Increase, Multiplicative Decrease: In this interval $[t_2, t_3]$, $W_n(t)$ multiplicatively decreases in response to observing queue lengths above q_t. However, the rate of decrease of $W_n(t)$ is bounded by the rate of increase of $W_r(t)$, as any faster a decrease will cause the queue size to drop below q_t. At the end of this interval $W(t_3) = \mu\tau + B$. At this point, each flow decreases its window size by a factor of γ, thereby entering into the next period.

In order to quantify the interference experienced by Reno flows because of the presence of Nice flows, we formulate differential equations to represent the variation of the queue size in a period. We then show that the values of W_r and W_n at the beginning of periods stabilize after several losses, so that the length of a period converges to a fixed value. It is then straightforward to compute the total amount of Reno flow sent out in a period. We show in the technical report [49] that the interference I, defined as the fractional loss in throughput experienced by Reno flows because of the presence of Nice flows, is given as follows.

Theorem 1: The interference I is given by

$$I \leq \frac{4m \cdot e^{(-\frac{B(1-t)\gamma}{m})}}{(\mu\tau + B)\gamma} \tag{1}$$

The derivation of I indicates that all three design features of Nice are fundamentally important for reducing interference. The interference falls exponentially with $B(1-t)$ or $B - q_t$, which reflects the time that Nice has to multiplicatively back off before packet losses occur. Intuitively, multiplicative decrease allows any number of Nice flows to get out of the way of additively increasing demand flows. The dependence on the ratio $\frac{B}{m}$ suggests that as the number of demand flows approaches the maximum queue size the non-interference property starts to break down. This breakdown is not surprising as each flow barely gets to maintain one packet in the queue and TCP Reno is known to behave anomalously under such circumstances [35]. In a well designed network, when $B \gg m$, it can be seen that the dependence on the threshold t is weak, *i.e.* interference is small when t is, and careful tuning of the exact value of t in this region is unnecessary. Our full analysis shows that the above bound on I holds even for the case when $m \ll l$. Allowing window sizes to multiplicatively decrease below one is crucial in this proof.

4 ns Controlled Tests

The goal of our simulation is to validate our hypotheses in a controlled environment. In particular, we wish to i) test the non-interference property of Nice and ii) determine if Nice gets any useful bandwidth for the workloads considered. By using controlled *ns* [36] simulations in this phase of the study we can stress test the system by varying network configurations and load to extreme values. We can also systematically compare the Nice algorithm against others. Overall, the experiments support our theses:

- Nice flows cause almost no interference irrespective of the number of flows.
- Nice gets a significant fraction of the available spare bandwidth.
- Nice performs better than other existing protocols, including Reno, Vegas, and Vegas with reduced α and β parameters.

4.1 Methodology

We use *ns* 2.1b8a for our simulation experiments. The topology used is a bar-bell in which N TCP senders transmit through a shared bottleneck link L to an equal number of receivers. The router connecting the senders to L becomes the bottleneck queue. Routers perform drop-tail FIFO queueing except in experiments with RED turned on. The buffer size is set to the bandwidth delay product. Packets are 512 bytes in size and the propagation delay is set to 50ms. We vary the capacity of the link in order to simulate different amounts of spare capacity.

We use a 15 minute section of a Squid proxy trace logged at UC Berkeley as the foreground traffic over L. The number of flows fluctuates as clients enter and leave the system as specified by the trace. On average there are

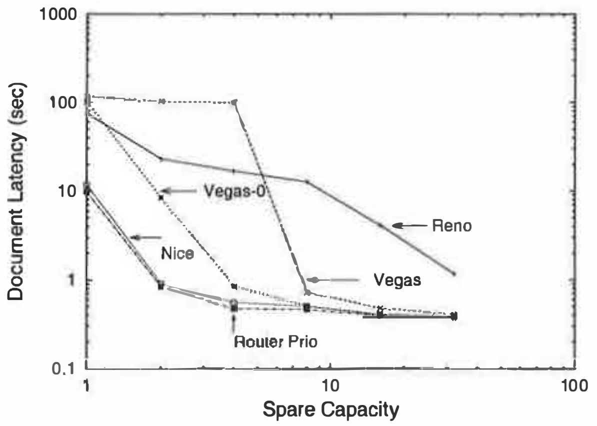


Figure 2: Spare capacity vs Latency

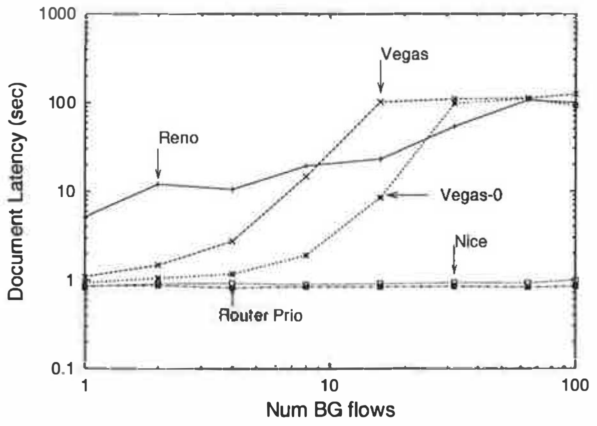


Figure 3: Number of BG flows vs Latency

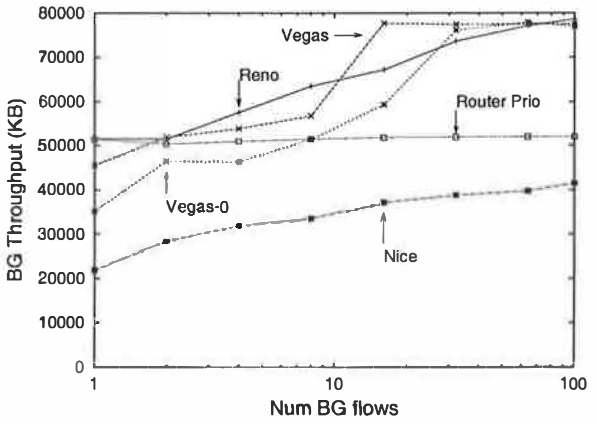


Figure 4: Number of BG flows vs BG throughput

about 12 active clients. In addition to this foreground load, we introduce permanently backlogged background flows. For the initial set of experiments we fix the bandwidth of the link to twice the average demand bandwidth of the trace. The primary metric we use to measure interference is the average transfer latency of a document *i.e.*, the time between its first packet being sent and the receipt of the ack corresponding to the last packet. We use the total number of bytes transferred by the background flows as the measure of its utilization of spare capacity.

Unless otherwise specified, the values of the $threshold$ and $fraction$ for Nice are set to 0.1 and 0.5 respectively. We compare the performance of Nice to several other strategies for sending background flows. First, we compare with router prioritization that services a background packet only if there are no queued foreground packets. Router prioritization is the ideal strategy for background flow transmission, as background flows never interfere with foreground flows. In addition, we compare to Reno, Vegas($\alpha = 1, \beta = 3$), Vegas($\alpha = 0, \beta = 0$).

4.2 Results

Experiment 1: In this experiment we fix the number of background flows to 16 and vary the spare capacity, S. To achieve a spare capacity S, we set the bottleneck link bandwidth $L = (1 + S) \cdot averageDemandBW$, where $averageDemandBW$ is the total number of bytes transferred in the trace divided by the duration of the trace. Figure 2 plots the average document transfer latency for foreground traffic as a function of the spare capacity in the network. Different lines represent different runs of the experiments using different protocols for background flows. It can be seen that Nice is hardly distinguishable from router prioritization whereas, the other protocols cause a significant increase in foreground latency. Note that the Y-axis is on a log scale, which means that in some cases Reno and Vegas increase foreground document transfer latencies by over an order of magnitude.

Experiment 2: Sensitivity to number of BG flows In this experiment we fix the spare capacity S of the network to 1 and vary the number of background flows. Figure 3 plots the latency of foreground document transfers against the number of background flows. Even with 100 background Nice flows, the latency of foreground documents is hardly distinguishable from the ideal case when routers provide strict prioritization. On the other hand, Reno and Vegas background flows can cause foreground latencies to increase by orders of magnitude. Figure 4 plots the number of bytes the background flows manage to transfer. A single background flow reaps about half the spare bandwidth available under router prioritization; this background throughput improves with increasing number of background flows but remains below router prioritization. The difference is the price we pay for ensuring non-interference with an end-to-end algorithm. Note that although Reno and Vegas obtain better throughput, even for a small number of flows they go beyond the router prioritization line, which means they steal bandwidth from foreground traffic.

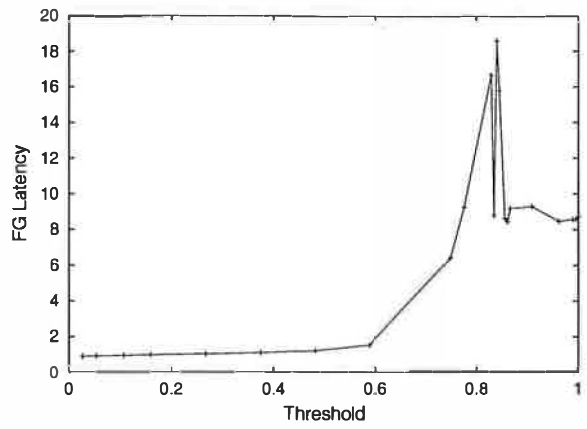


Figure 5: Threshold vs FG latency

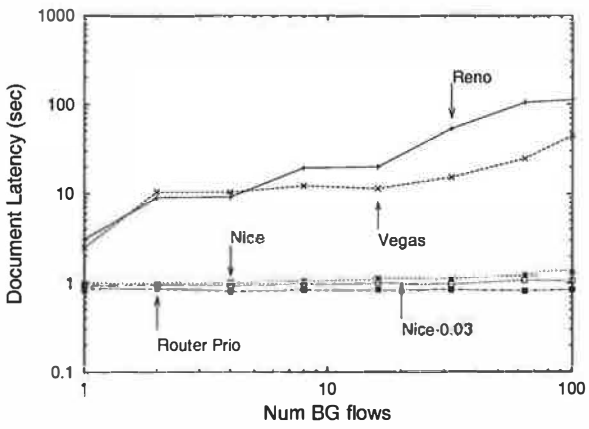


Figure 7: Number of BG flows vs Latency with RED

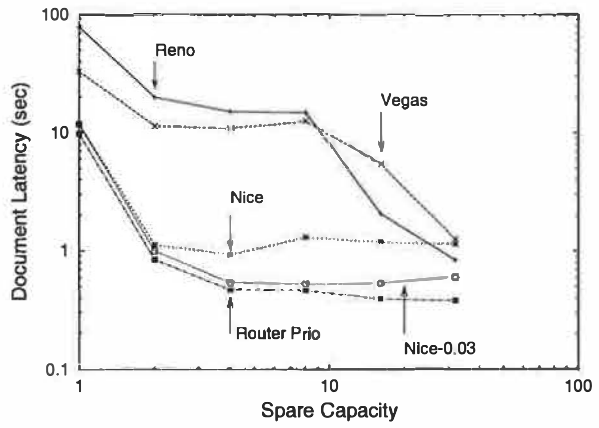


Figure 6: Spare capacity vs Latency with RED

We also examine experiments where we do not allow Nice's congestion window to fall below 1 (graph omitted). In this case, when the number of background flows exceeds about 10, the latency of foreground flows begins to increase noticeably; the increase is about a factor of two when the number of background flows is 64.

Experiment 3: Sensitivity to parameters In this experiment we trace the effect of the threshold and trigger fraction parameters described in Section 2.2. Figure 5 shows the document transfer latencies as a function of the threshold for the same trace as above, with $S = 1$ and 16 background flows. As expected, as the threshold value increases, the interference caused by Nice increases until the protocol finally reverts to Vegas behavior as the threshold approaches 1. It is interesting to note that there is large range of threshold values yielding low interference, which suggests that its value need not be manually tuned for each network. We examine the trigger fraction in the same way, and find little change in foreground latency as we vary this fraction from 0.1 to 0.9 (graph omitted).

Experiment 4: Nice with RED queueing We repeat experiments 1 and 2, but with routers performing RED queueing. The RED parameters are set as recommended in [23] with the "gentle" mode on. The minimum and maximum RED thresholds are set to one-third and two-third of the buffer size. Packets are probabilistically marked with ECN support from the senders. Figure 6 plots the foreground document transfer latency against the spare capacity with 16 background flows. Though Nice still performs as much as an order of magnitude better than other protocols, it causes up to a factor of 2 increase in document transfer latencies for large spare capacities. As figure 7 indicates, under RED, Nice closely approximates router prioritization regardless of the number of flows when the spare capacity is one, i.e. the demand workload consumes half of the network capacity.

The relatively poor performance of Nice under RED when spare capacities are large appears to reflect the sensitivity of Nice's interference I to bottleneck queue length (Equation 1). Whereas Nice flows damage foreground flows when drop-tail queues are completely full, under RED, interference can begin when the bottleneck queue occupancy reaches RED's minimum threshold min_{th}. One solution may be to reduce Nice's $threshold$ parameter. The *Nice-0.03* lines in Figures 6 and 7 plot Nice's interference under RED when $threshold = 0.03$ instead of the default value of 0.10. Future work is needed to better understand Nice's interaction with RED queuing.

Other results Due to space constraints, we state two other results here, but omit detailed discussions and graphs. The full discussion appears in the extended version [49].

First, we also perform experiments with synthetically generated ON/OFF Pareto UDP foreground traffic, which is much burstier and less predictable than TCP foreground flows. We observe that Nice still causes lower interference than Reno or Vegas, but does not match router prioritization as closely. The utilization of spare capacity by Nice is also lower compared to the trace workload case. This suggests that the benefits of Nice are reduced when traffic is unpredictable. Second, we compare Nice to simple rate limited Reno flows. When the rate is tuned to approximate the spare capacity of the network, rate limiting performs well. Nice, however, outperforms rate limiting and does not require hand tuning.

5 Internet Microbenchmarks

In this section we evaluate our Nice implementation over a variety of Internet links. We seek to answer three questions. First, in a less controlled environment than our NS simulations, does Nice still avoid interference? Second, are there enough reasonably long periods of spare capacity on real links for Nice to reap reasonable throughput? Third, are any such periods of spare capacity spread throughout the day, or is the usefulness of background transfers restricted to nights and weekends?

Our experiments suggest that Nice works for a range of networks, including a modem, a cable modem, a transatlantic link, and a fast WAN. In particular, on these networks it appears that Nice avoids interfering with other flows and that it can achieve throughput that are significant fractions of the throughput that would be achieved by Reno throughout the day.

5.1 Methodology

Our measurement client program connects to a measurement server program at exponentially-distributed random intervals. At each connection time, the client chooses one of six actions: Reno/NULL, Nice/NULL, Reno/Reno, Reno/Nice, Reno/Reno8, Reno/Nice8.[1] Each action consists of a "primary transfer" (denoted by the term left of the /) and zero or more "secondary transfers" (denoted by the term right of the /). Reno terms indicate flows using standard TCP-Reno congestion control. Nice terms indicate flows using Nice congestion control. For secondary transfers, NULL indicates actions that initiate no secondary transfers to compete with the primary transfer, and 8 indicates actions that initiate 8 (rather than the default 1) secondary transfers. The transfers are of large files whose sizes are chosen to require approximately 10 seconds for a single Reno flow to compete on the network under study.

[1]We also test standard Vegas in place of Reno for the large-transfer experiments and find that standard Vegas behaves essentially like Reno. These results are omitted due to space constraints.

We position a server that supports Nice at UT Austin. We position clients (1) in Austin connected to the Internet via a University of Texas 56.6K dial in modem bank (*modem*), (2) in Austin connected via a commercial ISP cable modem (*cable modem*), (3) in a commercial hosting center in London, England connected to multiple backbones including an OC12 and an OC3 to New York (*London*), and (4) at the University of Delaware, which connects to UT via an Abilene OC3 (*Delaware*). All machines run Linux. The server is a 450MHz Pentium II with 256MB of memory. The clients range from 450-1000MHz and all have at least 256MB of memory. The experiment ran from Saturday May 11 2002 to Wednesday May 15 2002; we gathered approximately 50 probes per client/workload pair.

5.2 Results

Figure 8 summarizes the results of our large-transfer experiments. On each of the networks, the throughput of Nice/NULL is a significant fraction of that of Reno/NULL, suggesting that periods of spare capacity are often long enough for Nice to detect and make use of them. Second, we note that during Reno/Nice and Reno/Nice8 actions, the primary (Reno) flow achieves similar throughput to the throughput seen during the control Reno/NULL sessions. In particular, on a modem network, when Reno flows compete with a single Nice flow, they receive on average 97% of the average bandwidth they receive when there is no competing Nice flow. On a cable modem network, when Reno flows compete with eight Nice flows, they receive 97% of the bandwidth they would receive alone. Conversely, Reno/Reno and Reno/Reno8 show the expected fair sharing of bandwidth among Reno flows, which reduces the bandwidth achieved by the primary flow.

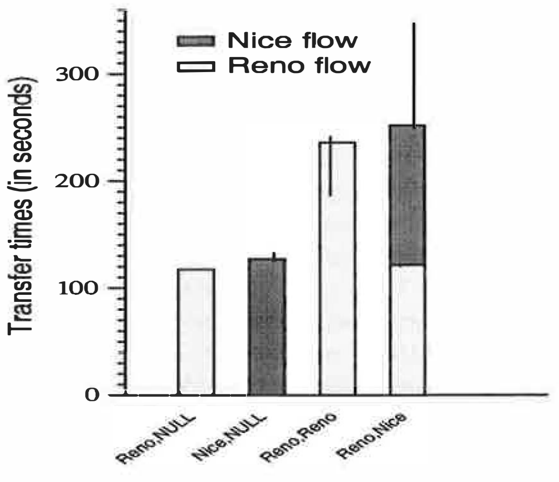
Figure 9 shows the hourly average bandwidth achieved by the primary flow for the different combinations listed above. Our hypothesis is that Nice can achieve useful amounts of throughput throughout the day, and the data appear to support this statement.

6 Case Study Applications

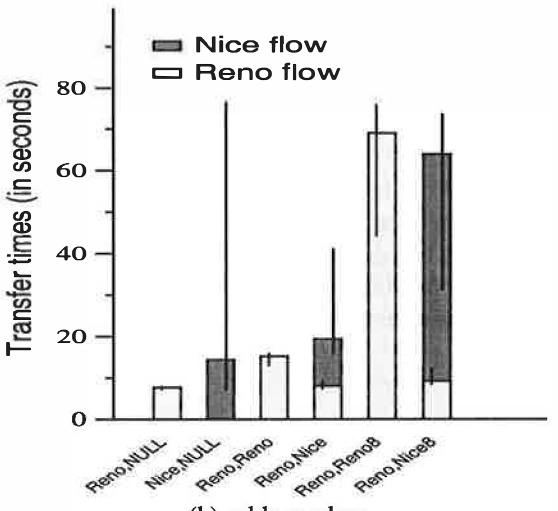
6.1 HTTP Prefetching

Many studies have published promising results that suggest that prefetching (or pushing) content could significantly improve web cache hit rates by reducing compulsory and consistency misses [15, 18, 26, 27, 32, 38, 50].

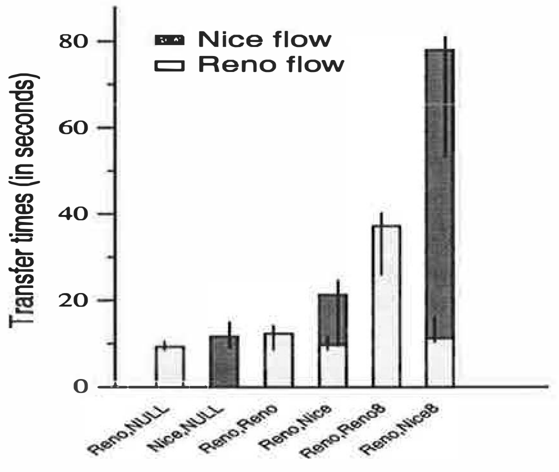
Typically, prefetching algorithms are tuned with a *threshold* parameter to balance the potential benefits of prefetching data against the bandwidth costs of fetching



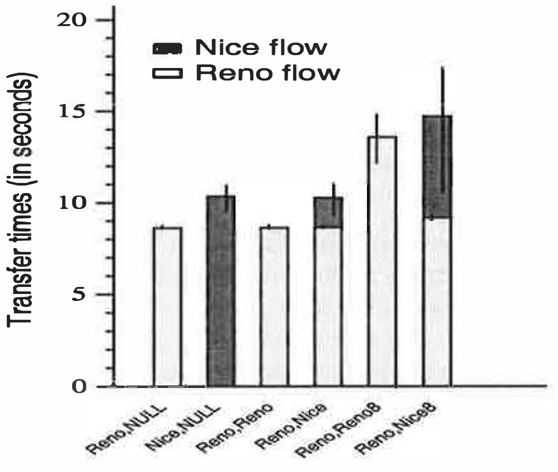
(a) modem



(b) cable modem



(c) London



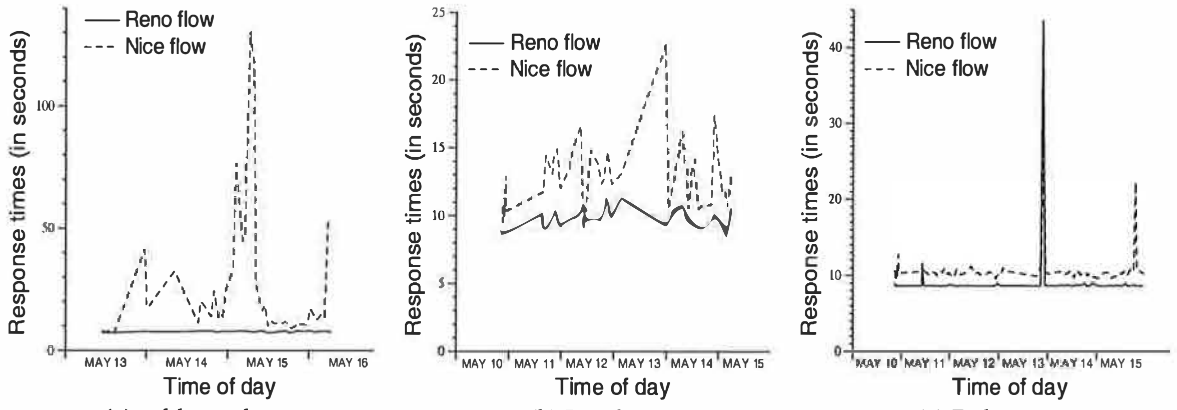
(d) Delaware

Figure 8: Large flow transfer performance. Each bar represents the average transfer time observed for the specified combination of primary/secondary transfers. Empty bars represent the average time for a Reno flow. Solid bars represent the average time for a Nice flow. The narrow lines depict the minimum and maximum values observed during multiple runs of each combination.

it and the storage cost of keeping it until its next use. An object is prefetched if the estimated probability that the object will be referenced before it is modified exceeds the threshold. Extending Gray and Shenoy's analysis of demand caching [25], Chandra calculates reasonable thresholds given network costs, disk costs, and human waiting time values and concludes that most algorithms in the literature have been far too conservative in setting their thresholds [9]. Furthermore, the 80-100% per year improvements in network [9, 37] and disk [17] capacity/cost mean that a value that is correct today may be off by an order of magnitude in 3-4 years.

In this case study, we build a prefetching protocol similar to the one proposed by Padmanabhan and Mogul [38]: when serving requests, servers piggy back lists of suggested objects in a new HTTP reply header. Clients receiving a prediction list discard old predictions and then issue prefetch requests of objects from the new list. This division of labor allows servers to use global information and application-specific knowledge to predict access patterns, and it allows clients to filter requests through their caches to avoid repeatedly fetching an object.

To evaluate prefetching performance, we implement a standalone client that reads a trace of HTTP requests, simulates a local cache, and issues demand and prefetch requests. Our client is written in Java and pipelines requests across HTTTP/1.1 persistent connections [21].



(a) cable modem (b) London (c) Delaware

Figure 9: Large flow transfer performance over time.

To ensure that demand and prefetch requests use separate TCP connections, our server directs prefetch requests to a different port than demand requests. The disadvantage of this approach is that it does not fit with the standard HTTP caching model. We discuss how to deploy such a protocol without modifying HTTP in a separate study [31].

We use Squid proxy traces from 9 regional proxies collected during January 2001 [51]. We study network interference near the server by examining subsets of the trace corresponding to a popular groups of related servers – *cnn* (e.g., cnn.com, www.cnn.com, cnnfn.com, etc.). This study compares relative performance for different resource management algorithms for a given set of prefetching algorithms. It does not try to identify optimal prefetching algorithms; nor does it attempt to precisely quantify the absolute improvements available from prefetching.

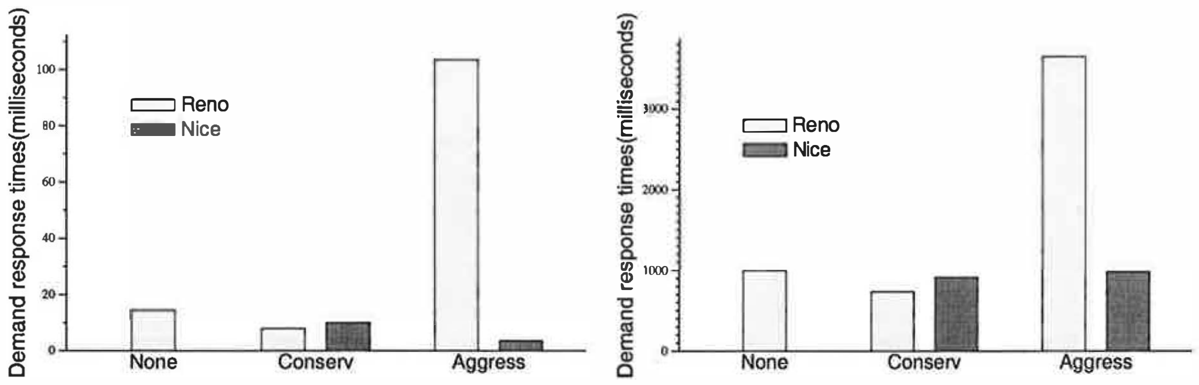
We use a simple prediction by partial matching algorithm [14] PPM-n/w that uses a client's n most recent requests to the server group for non-image data to predict cachable (i.e., non-dynamically-generated) URLs that will appear during a subsequent window that ends after the w'th non-image request to the server group. We use two variations of our PPM-n/w algorithm. The *conservative* variation uses parameters similar to those found in the literature for HTTP prefetching. It uses $n = 2$, $w = 5$ and sets the prefetch threshold to 0.25 [18]. To prevent prefetch requests from interfering with demand requests, it pauses 1 second after a demand reply is received before issuing requests. The *aggressive* variation uses $n = 2$, $w = 10$, and truncates prefetch proposal lists with a threshold probability of 0.00001. It issues prefetches immediately after receiving them.

We use 2 client machines connected to a server machine via a cable modem. On each client machine, we run 8 virtual clients, each with a separate cache and separate HTTP/1.1 demand and prefetch connections to the server. In order for the demand traffic to consume about 10% of the cable modem bandwidth, we select the 6 busiest hours from the 30-Jan-2001 trace and divide trace clients from each hour randomly across 4 of the virtual clients. In each of our seven trials, all the 16 virtual clients run the same prefetching algorithm: *none*, *conservative-Reno*, *aggressive-Reno*, *conservative-Nice*, *aggressive-Nice*.

Figure 10(a) shows the average demand response times perceived by the clients. We note that when clients do conservative prefetching using either protocol – Nice or Reno – the latency reductions are comparable. However, when they start aggressively prefetching using Reno, the latency blows up by an order of magnitude. Clients using aggressive Nice prefetching, however, continue to see further latency reductions. The figure shows that Nice is effective in using spare bandwidth for prefetching without affecting the demand requests.

Figure 10(b) represents the effect of prefetching over a modem (the setup is same as above except with the cable modem replaced by a modem), an environment where the amount of spare bandwidth available is minimal. This figure shows that while the Reno and Nice protocols are comparable in benefits when doing conservative prefetching, aggressive prefetching using Reno hurts the clients significantly by increasing the latencies three-fold. Nice on the other hand, does not worsen the latency even though it does not gain much.

We conclude that Nice simplifies the design of prefetching applications. Applications can aggressively prefetch data that might be accessed in the future. Nice prevents interference if the network does not have spare bandwidth and improves application performance if it does.



(a) cable modem (b) dial-up modem

Figure 10: Average demand transfer time for prefetching for the cnn server-group.

6.2 Tivoli Data Exchange

We study a simplified version of the Tivoli Data Exchange [20] system for replicating data across large numbers of hosts. This system distributes data and programs across thousands of client machines using a hierarchy of replication servers. Both non-interference and good throughput are important metrics. In particular, these data transfers should not interfere with interactive use of target machines. And because transfers may be large, may be time critical, and must go to a large number of clients using a modest number of simultaneous connections, each data transfer should complete as quickly as possible. The system currently uses two parameters at each replication server to tune the balance between non-interference and throughput. One parameter throttles the maximum rate that the server will send a single client; the other throttles the maximum total rate across all clients.

Choosing these rate limiting parameters requires some knowledge of network topology and may have to choose between overwhelming slow clients and slowing fast clients (e.g., distributing a 300MB Office application suite would nearly a day if throttled to use less than half a 56.6Kb/s modem). One could imagine a more complex system that allows the maximum bandwidth to be specified on a per-client basis, but such a system would be complex to configure and maintain.

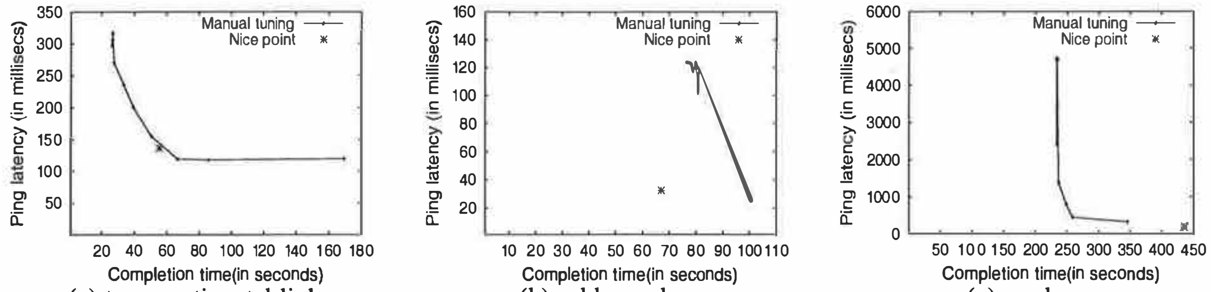
Nice can provide an attractive self-tuning abstraction. Using it, a sender can just send at the maximum speed allowed by the connection. We report preliminary results using a standalone server and client. The server and clients are the same as in the Internet measurements described in Section 5. We initiate large transfers from the server and during that transfer measure the ping round trip time between the client and the server. When running Reno, we vary the client throttle parameter and leave the total server bandwidth limit to an effectively infinite value. When running Nice, we set both the client and server bandwidth limits to effectively infinite values.

Figure 11 shows a plot of ping latencies (representative of interference) as a function of the completion time of transfers to clients over different networks. With Reno, completion times decrease with increasing throttle rates but increase ping latencies as well. Furthermore, the optimal rates vary widely across different networks. However Nice picks sending rates for each connection without the need for manual tuning that achieve minimum transfer times while maintaining acceptable ping latencies in all cases.

7 Related work

TCP congestion control has seen an enormous body of work since Jacobson's seminal paper on the topic [30]. This work seeks to maximize utilization of network capacity, to share the network fairly among flows, and to prevent pathological scenarios like congestion collapse. In contrast our primary goal is to ensure minimal interference with regular network traffic; though high utilization is important, it is a distinctly subordinate goal in our algorithm. Our algorithm is always less aggressive than AIMD TCP: it reacts the same way to losses and in addition, it reacts to increasing delays. Therefore, the work to ensure network stability under AIMD TCP applies to Nice as well.

The GAIMD [52] and binomial [4] frameworks provide generalized families of AIMD congestion control algorithms to allow protocols to trade smoothness for responsiveness in a TCP-friendly manner. The parameters can also be tuned to make a protocol less aggressive than TCP. We considered using these frameworks for constructing a background flow algorithm, but we were unable to develop the types of strong non-interference



(a) transcontinental link (b) cable modem (c) modem

Figure 11: Each continuous line represents completion times and corresponding ping latencies with varying send rates. The single point is the send rate chosen by Nice.

guarantees we seek using these frameworks. One area for future work is developing similar generalizations of Nice in order to allow different background flows to be more or less aggressive compared to one another while all remain completely timid with respect to competing foreground flows.

Prioritizing packet flows would be easier with router support. As noted in Section 4, router prioritization queues such as those proposed for DiffServ [5] service differentiation architectures are capable of completely isolating foreground flows from background flows while allowing background flows to consume nearly the entire available spare bandwidth. Unfortunately, these solutions are of limited use for someone trying to deploy a background replication service today because few applications are deployed solely in environments where router prioritization is installed or activated. A key conclusion of this study is that an end-to-end strategy need not rely on router support to make use of available network bandwidth without interfering with foreground flows.

Applications can limit the network interference they cause in various ways:
(a) *Coarse-grain scheduling*: Background transfers can be scheduled during hours where there is little foreground traffic. Studies [19, 34] show that prefetching data during off-peak hours can reduce latency and peak bandwidth usage.
(b) *Rate limiting*: Spring et. al [46] discuss prioritizing flows by controlling the receive window sizes of clients. Crovella et. al [15] propose a combination of window-based rate control and pacing to spread out prefetched traffic to limit interference. They show that such shaping of traffic leads to less bursty traffic and smaller queue lengths.
(c) *Application tuning*: Applications can limit the amount of data they send by varying application-level parameters. For example, many prefetching algorithms estimate the probability that an object will be referenced and only prefetch that object if its probability exceeds some threshold [18, 26, 38, 50].

It is not clear how an engineer should go about setting such application-specific parameters. We believe that self-tuning support for background transfers has at least three advantages over existing application-level approaches. Nice operates over fine time scales, so it can provide lower interference (by reacting to spikes in load) as well as higher average throughput (by using a large fraction of spare bandwidth) than static hand-tuned parameters. This property reduces the risk and increases the benefits available to background transfers while simplifying application design. Our experiments also demonstrate that Nice provides useful bandwidth throughout the day in many environments.

Existing transport layer solutions can be used to tackle the problem of self-interference between a single sender/receiver's flows. The congestion manager CM [3] provides an interface between the transport and the application layers to share information across connections and for handling applications using different transport protocols. Microsoft XP's Background Intelligent Transfer Service (BITS) provides support for transfers of lower priority to minimize interference with the user's interactive sessions by using a rate throttling approach. In contrast to these approaches, Nice handles both self- as well as cross-interference by modifying the sender side alone.

8 Conclusions

This paper presents an end-to-end congestion control algorithm optimized to support background transfers. Surprisingly, an end-to-end protocol can nearly approximate the ideal router-prioritization strategy by (a) almost eliminating interference with demand flows and (b) reaping significant fractions of available spare network bandwidth.

Our Internet experiments suggest that there is a significant amount of spare capacity on a wide variety of Internet links. Nice provides a mechanism to improve application performance by harnessing this capacity in a non-interfering manner. Our case studies demonstrate that Nice can simplify application design by eliminating the need to hand-tune parameters to balance utilization and interference. Inspired by the results in this paper, we have built a self-tuning prefetching system [31] based on Nice that avoids interference at the server and in the network, and is deployable with simple modifications to a web server.

One application of Nice is to support massive replication of data and services, where spare resources (e.g., bandwidth, disk space, and processor cycles) are consumed to help humans be more productive. Massive replication systems should be designed as if bandwidth were essentially free. TCP Nice provides a reasonable approximation of such an abstraction.

References

[1] Anurag Acharya and Joel Saltz. A study of internet round-trip delay. Technical Report CS-TR-3736, University of Maryland, 1996.

[2] Akamai, Inc. http://www.akamai.com.

[3] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System support for bandwidth management and content adaptation in internet applications. In *OSDI*, pages 213–226, 2000.

[4] D. Bansal and H. Balakrishnan. Binomial Congestion Control Algorithms. In *Infocom*, 2001.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services, 1998.

[6] T. Bonald. Comparision of TCP Reno and TCP Vegas via fluid approximation. INRIA Research Report 3563, Nov 1998.

[7] Lawrence S. Brakmo and Larry L. Peterson. TCP vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[8] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Infocom*, 1999.

[9] B. Chandra. Web workloads influencing disconnected service access. Master's thesis, University of Texas at Austin, May 2001.

[10] B. Chandra, M. Dahlin, L. Gao, A. Khoja, A. Razzaq, and A. Sewani. Resource management for scalable disconnected access to web services. In *WWW10*, May 2001.

[11] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end WAN Service Availability. In *USITS*, 2001.

[12] Chiu and Jain. Analysis of increase and decrease algorithms for congestion avoidance in computer networks. *Journal of Computer networks and ISDN*, 17(1):1–14, June 1989.

[13] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *SIGMOD*, 2000.

[14] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 1984.

[15] M. Crovella and P. Barford. The network effects of prefetching. In *Infocom*, 1998.

[16] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[17] M. Dahlin. http://www.cs.utexas.edu/users/dahlin/techTrends/data/diskPrices/data, Jan 2002.

[18] D. Duchamp. Prefetching Hyperlinks. In *USITS*, 1999.

[19] S. Dykes and K. A. Robbins. A viability analysis of cooperative proxy caching. In *Infocom*, 2001.

[20] Tivoli Data Exchange. http://www.tivoli.com/products/documents/datasheets/data_exchange_ds.pdf.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, June 1999.

[22] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications: the extended version. Technical Report TR-00-003, ICSI, March 2000.

[23] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[24] P. Goyal, X. Guo, and H.M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *OSDI*, pages 107–122, October 1996.

[25] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *"Proc. 16th Internat. Conference on Data Engineering"*, pages 3–12, 2000.

[26] J. Griffioen and R. Appleton. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.

[27] J. S. Gwertzman and M. Seltzer. The case for geographical push-caching. In *HotOS*, 1995.

[28] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[29] N. Hutchison, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. physical file system backup. In *OSDI*, 1999.

[30] V. Jacobson. Congestion avoidance and control. In *SIGCOMM*, 1988.

[31] R. Kokku, P. Yalagandula, A. Venkataramani, and M. Dahlin. A non-interfering deployable web prefetching system. Technical Report TR-02-51, Computer Sciences, UT Austin, May 2002.

[32] T. M. Kroeger, D. E. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USITS*, 1997.

[33] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *OSDI*, 2000.

[34] C. Maltzahn, K. Richardson, D. Grunwald, and J. Martin. On bandwidth smoothing. In *4th International Web Caching Workshop*, 1999.

[35] R. Morris. Tcp behavior with many flows. In *International Conference on Network Protocols*, 1997.

[36] The network simulator – ns-2. http://www.isi.edu/nsnam/ns.

[37] A. Odlyzko. Internet growth: Myth and reality, use and abuse. *Journal of Computer Resource Management*, pages 23–27, 2001.

[38] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve World-Wide Web latency. In *SIGCOMM*, 1996.

[39] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *SOSP*, 1995.

[40] V. Paxson. End-to-end Routing Behavior in the Internet. In *SIGCOMM*, 1996.

[41] G. Popek, R. Guy, T. Page, and J. Heidemann. Replication in the Ficus Distributed File System. In *Workshop on the Management of Replicated Data*, pages 5–10, November 1990.

[42] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet. In *Infocom*, 1999.

[43] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.

[44] Dheeraj Sanghi, Ashok K. Agrawala, Olafur Gudmundsson, and Bijendra N. Jain. Experimental assessment of end-to-end behavior on internet. In *Infocom (2)*, pages 867–874, 1993.

[45] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next-generation operating systems. In *SIGMETRICS*, 1998.

[46] Neil T. Spring, Maureen Chesire, Mark Berryman, Vivek Sahasranaman, Thomas Anderson, and Brian N. Bershad. Receiver based management of low bandwidth access links. In *Infocom*, 2000.

[47] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*, 1995.

[48] A. Venkataramani, M. Dahlin, and P. Weidmann. Bandwidth constrained placement in a WAN. In *PODC*, 2001.

[49] A. Venkataramani, R. Kokku, and M. Dahlin. System support for background replication. Technical Report TR-02-30, Computer Sciences, UT Austin, May 2002.

[50] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. Potential costs and benefits of long-term prefetching for content-distribution. *Computer Communications Journal*, 25(4):367–375, 2002.

[51] D. Wessels. Squid Internet object cache. http://squid.nlanr.net/Squid, Jan 1998.

[52] Y. Yang and S. Lam. General AIMD Congestion Control. In *ICNP*, 2000.

[53] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *SOSP*, 2001.

[54] Y. Zhang, V. Paxson, and S. Shenkar. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. Technical report, ICSI Center for Internet Research, May 2000.

The Effectiveness of Request Redirection on CDN Robustness

Limin Wang, Vivek Pai and Larry Peterson
Department of Computer Science
Princeton University
{lmwang,vivek,llp}@cs.princeton.edu

Abstract

It is becoming increasingly common to construct network services using redundant resources geographically distributed across the Internet. Content Distribution Networks are a prime example. Such systems distribute client requests to an appropriate server based on a variety of factors---e.g., server load, network proximity, cache locality—in an effort to reduce response time and increase the system capacity under load. This paper explores the design space of strategies employed to redirect requests, and defines a class of new algorithms that carefully balance load, locality, and proximity. We use large-scale detailed simulations to evaluate the various strategies. These simulations clearly demonstrate the effectiveness of our new algorithms, which yield a 60-91% improvement in system capacity when compared with the best published CDN technology, yet user-perceived response latency remains low and the system scales well with the number of servers.

1 Introduction

As the Internet becomes more integrated into our everyday lives, the availability of information services built on top of it becomes increasingly important. However, overloaded servers and congested networks present challenges to maintaining high accessibility. To alleviate these bottlenecks, it is becoming increasingly common to construct network services using redundant resources, so-called Content Distribution Networks (CDN) [1, 13, 21]. CDNs deploy geographically-dispersed server surrogates and distribute client requests to an "appropriate" server based on various considerations.

CDNs are designed to improve two performance metrics: *response time* and *system throughput*. Response time, usually reported as a cumulative distribution of latencies, is of obvious importance to clients, and represents the primary marketing case for CDNs. System throughput, the average number of requests that can be satisfied each second, is primarily an issue when the system is heavily loaded, for example, when a flash crowd is accessing a small set of pages, or a Distributed Denial of Service (DDoS) attacker is targeting a particular site [15]. System throughput represents the overall robustness of the system since either a flash crowd or a DDoS attack can make portions of the information space inaccessible.

Given a sufficiently widespread distribution of servers, CDNs use several, sometimes conflicting, factors to decide how to distribute client requests. For example, to minimize response time, a server might be selected based on its *network proximity*. In contrast, to improve the overall system throughput, it is desirable to evenly *balance* the load across a set of servers. Both throughput and response time are improved if the distribution mechanism takes *locality* into consideration by selecting a server that is likely to already have the page being requested in its cache.

Although the exact combination of factors employed by commercial systems is not clearly defined in the literature, evidence suggests that the scale is tipped in favor of reducing response time. This paper addresses the problem of designing a request distribution mechanism that is both responsive across a wide range of loads, and robust in the face of flash crowds and DDoS attacks. Specifically, our main contribution is to explore the design space of strategies employed by the request redirectors, and to define a class of new algorithms that carefully balance load, locality, and proximity. We use large-scale detailed simulations to evaluate the various strategies. These simulations clearly demonstrate the effectiveness of our new algorithms: they produce a 60-91% improvement in system capacity when compared with published information about commercial CDN technology, user-perceived response latency remains low, and the system scales well with the number of servers. We also discuss several implementation issues, but evaluating a specific implementation is beyond the scope of this paper.

2 Building Blocks

The idea of a CDN is to geographically distribute a collection of *server surrogates* that cache pages normally maintained in some set of *backend servers*. Thus, rather than let every client try to connect to the original server, it is possible to spread request load across many servers. Moreover, if a server surrogate happens to reside close to the client, the client's request could be served without having to cross a long network path. In this paper, we observe this general model of a CDN, and assume any of the server surrogates can serve any request on behalf of the original server. Where to place these surrogates, and how to keep their contents up-to-date, has been addressed by other CDN research [1, 13, 21]. Here, we make no particular assumptions about servers' strategic locations.

Besides a large set of servers, CDNs also need to provide a set of *request redirectors*, which are middleware entities that forward client requests to appropriate servers based on one of the strategies described in the next section. To help understand these strategies, this section first outlines various mechanisms that could be employed to implement redirectors, and then presents a set of hashing schemes that are at the heart of redirection.

2.1 Redirector Mechanisms

Several mechanisms can be used to redirect requests [3], including augmented DNS servers, HTTP-based redirects, and smart intermediaries such as routers or proxies.

A popular redirection mechanism used by current CDNs is to augment DNS servers to return different server addresses to clients. Without URL rewriting that changes embedded objects to point to different servers, this approach has site-level granularity, while schemes that rewrite URLs can use finer granularity and thus spread load more evenly. Client-side caching of DNS mappings can be avoided using short expiration times.

Servers can perform the redirection process themselves by employing the HTTP "redirect" response. However, this approach incurs an additional round-trip time, and leaves the servers vulnerable to overload by the redirection task itself. Server bandwidth is also consumed by this process.

The redirection function can also be distributed across intermediate nodes of the network, such as routers or proxies. These redirectors either rewrite the outbound requests, or send HTTP redirect messages back to the client. If the client is not using explicit (forward mode) proxying, then the redirectors must be placed at choke points to ensure traffic in both forward and reverse directions is handled. Placing proxies closer to the edge yields well-confined easily-identifiable client populations, while moving them closer to the server can result in more accurate feedback and load information.

To allow us to focus on redirection strategies and to reduce the complexity of considering the various combinations outlined in this section, we make the following assumptions: redirectors are located at the edge of a client site, they receive the full list of server surrogates through DNS or some other out-of-band communication, they rewrite outbound requests to pick the appropriate server, and they passively learn approximate server load information by observing client communications. We do not rely on any centralization, and all redirectors operate independently. Our experiments show that these assumptions—in particular, the imperfect information about server load—do not have a significant impact on the results.

2.2 Hashing Schemes

Our geographically dispersed redirectors cannot easily adapt the request routing schemes suited for more tightly-coupled LAN environments [17, 25], since the latter can easily obtain instantaneous state about the entire system. Instead, we construct strategies that use hashing to deterministically map URLs into a small range of values. The main benefit of this approach is that it eliminates inter-redirector communication, since the same output is produced regardless of which redirector receives the URL. The second benefit is that the range of resulting hash values can be controlled, trading precision for the amount of memory used by bookkeeping.

The choice of which hashing style to use is one component of the design space, and is somewhat flexible. The various hashing schemes have some impact on computational time and request reassignment behavior on node failure/overload. However, as we discuss in the next section, the computational requirements of the various schemes can be reduced by caching.

Modulo Hashing – In this "classic" approach, the URL is hashed to a number modulo the number of servers. While this approach is computationally efficient, it is unsuitable because the modulus changes when the server set changes, causing most documents to change server assignments. While we do not expect frequent changes in the set of servers, the fact that the addition of new servers into the set will cause massive reassignment is undesirable.

Consistent Hashing [19, 20] – In this approach, the URL is hashed to a number in a large, circular space, as are the names of the servers. The URL is assigned to the server that lies closest on the circle to its hash value. A search tree can be used to reduce the search to logarithmic time. If a server node fails in this scheme, its load shifts to its neighbors, so the addition/removal of a server only causes local changes in request assignments.

Highest Random Weight [31] – This approach is the basis for CARP [8], and consists of generating a list of hash values by hashing the URL and each server's name and sorting the results. Each URL then has a deterministic order to access the set of servers, and this list is traversed until a suitably-loaded server is found. This approach requires more computation than Consistent Hashing, but has the benefit that each URL has a different server order, so a server failure results in the remaining servers evenly sharing the load. To reduce computation cost, the top few entries for each hash value can be cached.

3 Strategies

This section explores the design space for the request redirection strategies. As a quick reference, we summarize the properties of the different redirection algorithms in Table 1, where the strategies are categorized based on how they address locality, load and proximity.

Category	Strategy	Hashing Scheme	Dynamic Server Set	Load Aware
Random	Random			No
Static	R-CHash	CHash	No	No
	R-HRW	HRW	No	No
Static +Load	LR-CHash	CHash	No	Yes
	LR-HRW	HRW	No	Yes
Dynamic	CDR	HRW	Yes	Yes
	FDR	HRW	Yes	Yes
	FDR-Global	HRW	Yes	Yes
Network Proximity	NPR-CHash	CHash	No	No
	NPLR-CHash	CHash	No	Yes
	NP-FDR	HRW	Yes	Yes

Table 1: Properties of Request Redirection Strategies

The first category, Random, contains a single strategy, and is used primarily as a baseline. We then discuss four static algorithms, in which each URL is mapped onto a fixed set of server replicas—the Static category includes two schemes based on the best-known published algorithms, and the Static+Load category contains two variants that are aware of each replica's load. The four algorithms in these two static categories pay increasing attention to locality. Next, we introduce two new algorithms—denoted **CDR** and **FDR**—that factor both load and locality into their decision, and each URL is mapped onto a dynamic set of server replicas. We call this the Dynamic category. Finally, we factor network proximity into the equation, and present another new algorithm—denoted **NP-FDR**—that considers all aspects of network proximity, server locality, and load.

3.1 Random

In the random policy, each request is randomly sent to one of the server surrogates. We use this scheme as a baseline to determine a reasonable level of performance, since we expect the approach to scale with the number of servers and to not exhibit any pathological behavior due to patterns in the assignment. It has the drawback that adding more servers does not reduce the working set of each server. Since serving requests from main memory is faster than disk access, this approach is at a disadvantage versus schemes that exploit URL locality.

3.2 Static Server Set

We now consider a set of strategies that assign a fixed number of server replicas to each URL. This has the effect of improving locality over the Random strategy.

3.2.1 Replicated Consistent Hashing

In the Replicated Consistent Hashing (**R-CHash**) strategy, each URL is assigned to a set of replicated servers. The number of replicas is fixed, but configurable. The URL is hashed to a value in the circular space, and the replicas are evenly spaced starting from this original point. On each request, the redirector randomly assigns the request to one of the replicas for the URL. This strategy is intended to model the mechanism used in published content distribution networks, and is virtually identical[1] to the scheme described in [19] and [20] with the network treated as a single geographic region.

3.2.2 Replicated Highest Random Weight

The Replicated Highest Random Weight (**R-HRW**) strategy is the counterpart to R-CHash, but with a different hashing scheme used to determine the replicas. To the best of our knowledge, this approach is not used in any existing content distribution network. In this approach, the set of replicas for each URL is determined by using the top N servers from the ordered list generated by Highest Random Weight hashing. Versus R-CHash, this scheme is less likely to generate the same set of replicas for two different URLs. As a result, the less-popular URLs that may have some overlapping servers with popular URLs are also likely to have some other less-loaded nodes in their replica sets.

3.3 Load-Aware Static Server Set

The Static Server Set schemes randomly distribute requests across a set of replicas, which shares the load but without any active monitoring. We extend these schemes by introducing load-aware variants of these approaches. To perform fine-grained load balancing, these schemes maintain local estimates of server load at the redirectors, and use this information to pick the least-loaded member of the server set. The load-balanced variant of R-CHash is called **LR-CHash**, while the counterpart for R-HRW is called **LR-HRW**.

3.4 Dynamic Server Set

We now consider a new category of algorithms that dynamically adjust the number of replicas used for each URL in an attempt to maintain both good server locality and load balancing. By reducing unnecessary replication, the working set of each server is reduced, resulting in better file system caching behavior.

[1]The scheme described in these papers also includes a mechanism to use coarse-grained load balancing via virtual server names. When server overload is detected, the corresponding content is replicated across all servers in the region, and the degree of replication shrinks over time. However, the schemes are not described in enough detail to replicate.

3.4.1 Coarse Dynamic Replication

Coarse Dynamic Replication (**CDR**) adjusts the number of replicas used by redirectors in response to server load and demand for each URL. Like R-HRW, CDR uses HRW hashing to generate an ordered list of servers. Rather than using a fixed number of replicas, however, the request target is chosen using coarse-grained server load information to select the first "available" server on the list.

Figure 1 shows how a request redirector picks the destination server for each request. This decision process is done at each redirector independently, using the load status of the possible servers. Instead of relying on heavy communications between servers and request redirectors to get server load status, we use local load information observed by each redirector as an approximation. We currently use the number of active connections to infer the load level, but we can also combine this information with response latency, bandwidth consumption, etc.

```
find_server($url$, $S$) {
  foreach server $s_i$ in server set $S$,
    $weight_i$ = hash($url$, address($s_i$));
  sort $weight$;
  foreach server $s_j$ in decreasing order of $weight_j$ {
    if $satisfy\_load\_criteria(s_j)$ then {
      $targetServer \leftarrow s_j$;
      stop search;
    }
  }
  if $targetServer$ is not valid then
    $targetServer \leftarrow$ server with highest weight;
  route request $url$ to $targetServer$;
}
```

Figure 1: Coarse Dynamic Replication

As the load increases, this scheme changes from using only the first server on the sorted list to spreading requests across several servers. Some documents normally handled by "busy" servers will also start being handled by less busy servers. Since this process is based on aggregate server load rather than the popularity of individual documents, servers hosting some popular documents may find more servers sharing their load than servers hosting collectively unpopular documents. In the process, some unpopular documents will be replicated in the system simply because they happen to be primarily hosted on busy servers. At the same time, if some documents become extremely popular, it is conceivable that all of the servers in the system could be responsible for serving them.

3.4.2 Fine Dynamic Replication

A second dynamic algorithm—Fine Dynamic Replication (**FDR**)—addresses the problem of unnecessary replication in CDR by keeping information on URL popularity and using it to more precisely adjust the number of replicas. By controlling the replication process, the per-server working sets should be reduced, leading to better server locality, and thereby better response time and throughput.

The introduction of finer-grained bookkeeping is an attempt to counter the possibility of a "ripple effect" in CDR, which could gradually reduce the system to round-robin under heavy load. In this scenario, a very popular URL causes its primary server to become overloaded, causing extra load on other machines. Those machines, in turn, also become overloaded, causing documents destined for them to be served by their secondary servers. Under heavy load, it is conceivable that this displacement process ripples through the system, reducing or eliminating the intended locality benefits of this approach.

```
find_server($url$, $S$) {
  walk_entry $\leftarrow$ walkLenHash($url$);
  $w\_len \leftarrow$ walk_entry.length;
  foreach server $s_i$ in server set $S$,
    $weight_i$ = hash($url$, address($s_i$));
  sort $weight$;
  $s_{candidate} \leftarrow$ least-loaded server of top $w\_len$ servers;
  if $satisfy\_load\_criteria(s_{candidate})$ then {
    $targetServer \leftarrow s_{candidate}$;
    if ($w\_len > 1$ &&
      timenow() $-$ walk_entry.lastUpd $> chgThresh$)
      walk_entry.length $--$;
  } else {
    foreach rest server $s_j$ in decreasing weight order {
      if $satisfy\_load\_criteria(s_j)$ then {
        $targetServer \leftarrow s_j$;
        stop search;
      }
    }
    walk_entry.length $\leftarrow$ actual search steps;
  }
  if walk_entry.length changed then
    walk_entry.lastUpd $\leftarrow$ timenow();
  if $targetServer$ is not valid then
    $targetServer \leftarrow$ server with highest weight;
  route request $url$ to $targetServer$;
}
```

Figure 2: Fine Dynamic Replication

To reduce extra replication, FDR keeps an auxiliary structure at each redirector that maps each URL to a "walk length," indicating how many servers in the HRW list should be used for this URL. Using a minimum walk length of one provides minimal replication for most URLs, while using a higher minimum will always distribute URLs over multiple servers. When the redirector receives a request, it uses the current walk length for the URL and picks the least-loaded server from the current set. If even this server is busy, the walk length is increased and the least-loaded server is used.

This approach tries to keep popular URLs from overloading servers and displacing less-popular objects in the process. The size of the auxiliary structure is capped by hashing the URL into a range in the thousands to millions. Hash collisions may cause some URLs to have their replication policies affected by popular URLs. As long as the the number of hash values exceeds the number of servers, the granularity will be significantly better than the Coarse Dynamic Replication approach. The redirector logic for this approach is shown in Figure 2. To handle URLs that become less popular over time, with each walk length, we also keep the time of its last modification. We decrease the walk length if it has not changed in some period of time.

As a final note, both dynamic replication approaches require some information about server load, specifically how many outstanding requests can be sent to a server by a redirector before the redirector believes it is busy. We currently allow the redirectors to have 300 outstanding requests per server, at which point the redirector locally decides the server is busy. It would also be possible to calibrate these values using both local and global information—using its own request traffic, the redirector can adjust its view of what constitutes heavy load, and it can perform opportunistic communication with other redirectors to see what sort of collective loads are being generated. The count of outstanding requests already has some feedback, in the sense that if a server becomes slow due to its resources (CPU, disk, bandwidth, etc.) being stressed, it will respond more slowly, increasing the number of outstanding connections. To account for the inaccuracy of local approximation of server load at each redirector, in our evaluations, we also include a reference strategy, **FDR-Global**, where all redirectors have perfect knowledge of the load at all servers.

Conceivably, Consistent Hashing could also be used to implement CDR and FDR. We tested a CHash-based CDR and FDR, but they suffer from the "ripple effect" and sometimes yield even worse performance than load-aware static replication schemes. Part of the reason is that in Consistent Hashing, since servers are mapped onto a circular space, the relative order of servers for each URL will be *effectively* the same. This means the load migration will take an uniform pattern; and the less-popular URLs that may have overlapping servers with popular URLs are unlikely to have some other less-loaded nodes in their replica sets. Therefore, in this paper, we will only present CDR and FDR based on HRW.

3.5 Network Proximity

Many commercial CDNs start server selection with network proximity matching. For instance, [19] indicates that CDN's hierarchical authoritative DNS servers can map a client's (actually its local DNS server's) IP address to a geographic region within a particular network and then combine it with network and server load information to select a server separately within each region. Other research [18] shows that in practice, CDNs succeed not by always choosing the "optimal" server, but by avoiding notably bad servers.

For the sake of studying system capacity, we make a conservative simplifcaction by treating the entire network topology as a single geographic region. We could also simply take the hierarchical region approach as in [19], however, to see the effect of *integrating* proximity into server selection, we introduce three strategies that explicitly factor intra-region network proximity into the decision. Our redirector measures servers' geographical/topological location information through *ping*, *traceroute* or similiar mechanisms and uses this information to calculate an "effective load" when choosing servers.

To calculate the *effective load*, redirectors multiply the raw load metric with a normalized *standard distance* between the redirector and the server. Redirectors gather distances to servers using round trip time (RTT), routing hops, or similar information. These raw distances are normalized by dividing by the minimum locally-observed distance, yielding the standard distance. In our simulations, we use RTT for calculating raw distances.

FDR with Network Proximity (NP-FDR) is the counterpart of FDR, but it uses effective load rather than raw load. Similarly, **NPLR-CHash** is the proximity-aware version of LR-CHash. The third strategy, **NPR-CHash**, adds network proximity to the load-oblivious R-CHash approach by assigning requests such that each surrogate in the fixed-size server set of a URL will get a share of total requests for that URL inversely proportional to the surrogate's distance from the redirector. As a result, closer servers in the set get a larger share of the load.

The use of effective load biases server selection in favor of closer servers when raw load values are comparable. For example, in standard FDR, raw load values reflect the fact that distant servers generate replies more slowly, so some implicit biasing exists. However, by explicitly factoring in proximity, NP-FDR attempts to reduce global resource consumption by favoring shorter network journeys.

Although we currently calculate effective load this way, other options exist. For example, effective load can take other dynamic load/proximity metrics into account, such as network congestion status through real time measurement, thereby reflecting instantaneous load conditions.

4 Evaluation Methodology

The goal of this work is to examine how these strategies respond under different loads, and especially how robust they are in the face of flash crowds and other abnormal workloads that might be used for a DDoS attack. Attacks may take the form of legitimate traffic, making them difficult to distinguish from flash crowds.

Evaluating the various algorithms described in Section 3 on the Internet is not practical, both due to the scale of the experiment required and the impact a flash crowd or attack is likely to have on regular users. Simulation is clearly the only option. Unfortunately, there has not been (up to this point) a simulator that considers both network traffic and server load. Existing simulators either focus on the network, assuming a constant processing cost at the server, or they accurately model server processing (including the cache replacement strategy), but use a static estimate for the network transfer time. In the situations that interest us, both the network and the server are important.

To remedy this situation, we developed a new simulator that combines network-level simulation with OS/server simulation. Specifically, we combine the NS simulator with Logsim, allowing us to simulate network bottlenecks, round-trip delays, and OS/server performance. NS-2 [23] is a packet-level simulator that has been widely-used to test TCP implementations. However, it does not simulate much server-side behavior. Logsim is a server cluster simulator used in previous research on LARD [25], and it provides detailed and accurate simulation of server CPU processing, memory usage, and disk access. This section describes how we combine these two simulators, and discusses how we configure the resulting simulator to study the algorithms presented in Section 3.

4.1 Simulator

A model of Logsim is shown in Figure 3. Each server node consists of a CPU and locally attached disk(s), with separate queues for each. At the same time, each server node maintains its own memory cache of a configurable size and replacement policy. Incoming requests are first put into the holding queue, and then moved to the active queue. The active queue models the parallelism of the server, for example, in multiple process or thread server systems, the maximum number of processes or threads allowed on each server.

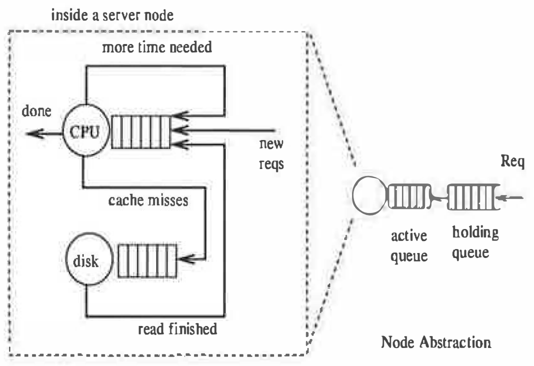


Figure 3: Logsim Simulator

We combined Logsim with NS-2 as follows. We keep NS-2's event engine as the main event manager, wrap each Logsim event as a NS-2 event, and insert it into the NS-2 event queue. All the callback functions are kept unchanged in Logsim. When crossing the boundary between the two simulators, tokens (continuations) are used to carry side-specific information. To speed up the simulation time, we also re-implemented several NS-2 modules and performed other optimizations.

On the NS-2 side, all packets are stored and forwarded, as in a real network, and we use two-way TCP. We currently use static routing within NS-2, although we may run simulations with dynamic routing in the future.

On the Logsim side, the costs for the basic request processing were derived by performing measurements on a 300MHz Pentium II machine running FreeBSD 2.2.5 and the Flash web server [24]. Connection establishment and tear-down costs are set at $145\mu s$, while transmit processing incurs $40\mu s$ per 512 bytes. Using these numbers, an 8-KByte document can be served from the main memory cache at a rate of approximately 1075 requests/sec. When disk access is needed, reading a file from the disk has a latency of 28ms. The disk transfer time is $410\mu s$ per 4 KBytes. For files larger than 44 KBytes, and additional 14ms is charged for every 44 KBytes of file length in excess of 44 KBytes. The replacement policy used on the servers is Greedy-Dual-Size (GDS)[5], as it appears to be the best known policy for Web workloads. 32MB memory is available for caching documents on each server and every server node has one disk. This server is intentionally slower than the current state-of-the-art (it is able to service approximately 600 requests per second), but this allows the simulation to scale to a larger number of nodes.

The final simulations are very heavy-weight, with over a thousand nodes and a very high aggregate request rate. We run the simulator on a 4-processor/667MHz Alpha with 8GB RAM. Each simulation requires 2-6GB of RAM, and generally takes 20-50 hours of wall-clock time.

4.2 Network Topology

It is not easy to find a topology that is both realistic and makes the simulation manageable. We choose to use a slightly modified version the NSFNET backbone network T3 topology, as shown in Figure 4.

In this topology, the round-cornered boxes represent backbone routers with the approximate geographical location label on it. The circles, tagged as R1, R2..., are regional routers;[2] small circles with "C" stand for client hosts; and shaded circles with "S" are the server surrogates. In the particular configuration shown in the figure, we put 64 servers behind regional routers R0, R1, R7, R8, R9, R10, R15, R19, where each router sits in front of 8 servers. We distribute 1,000 client hosts evenly behind the other regional routers, yielding a topology of nearly 1,100 nodes. The redirector algorithms run on the regional routers that sit in front of the clients.

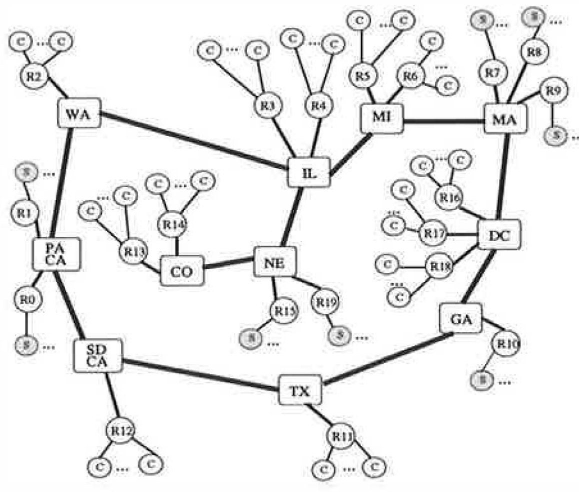


Figure 4: Network Topology

The latencies of servers to regional routers are set randomly between 1ms to 3ms; those of clients to regional routers are between 5ms and 20ms; those of regional routers to backbone routers are between 1 to 10ms; latencies between backbone routers are set roughly according to their geographical distances, ranging from 8ms to 28ms.

To simulate high request volume, we deliberately provision the network with high link bandwidth by setting the backbone links at 2,488Mbps, and links between regional routers and backbone routers at 622Mbps. Links between servers and regional routers are 100Mbps and those between clients and their regional servers are randomly between 10Mbps and 45Mbps. All the queues at routers are drop tail, with the backbone routers having room to buffer 1024 packets, and all other routers able to buffer 512 packets.

[2]These can also be thought of as edge/site routers, or the boundary to an autonomous system

4.3 Workload and Stability

We determine system capacity using a trace-driven simulation and gradually increase the aggregate request rate until the system fails. We use a two month trace of server logs obtained at Rice University, which contains 2.3 million requests for 37,703 files with a total size of 1,418MB [25], and has properties similar to other published traces.

The simulation starts with the clients sharing the trace and sending requests at a low aggregate rate in an open-queue model. Each client gets the name of the document sequentially from the shared trace when it needs to send a request, and the timing information in the trace is ignored. The request rate is increased by 1% every simulated six seconds, regardless of whether previous requests have completed. This approach gradually warms the server memory caches and drives the servers to their limits over time. We configure Logsim to handle at most 512 simultaneous requests and queue the rest. The simulation is terminated when the offered load overwhelms the servers.

Flash crowds, or DDoS attacks in bursty legitimate traffic form, are simulated by randomly selecting some clients as *intensive* requesters and randomly picking a certain number of hot-spot documents. These intensive requesters randomly request the hot documents at the same rate as normal clients, making them look no different than other legitimate users. We believe that this random distribution of intensive requesters and hot documents is a quite general assumption since we do not require any special detection or manual intervention to signal the start of a flash crowd or DDoS attack.

We define a server as being overloaded when it can no longer satisfy the rate of incoming requests and is unlikely to recover in the future. This approach is designed to determine when service is actually being denied to clients, and to ignore any short-term behavior which may be only undesirable, rather than fatal. Through experimentation, we find that when a server's request queue grows beyond 4 to 5 times the number of simultaneous connections it can handle, throughput drops and the server is unlikely to recover. Thus, we define the threshold for a server *failure* to be when the request queue length exceeds five times the simultaneous connection parameter. Since we increase the offered load 1% every 6 seconds, we record the request load exactly 30 seconds before the first server fails, and declare this to be the system's maximum capacity.

Although we regard any single server failure as a system failure in our simulation, the strategies we evaluate all exhibit similar behavior—significant numbers of servers fail at the same time, implying that our approach to deciding system capacity is not biased toward any particular scheme.

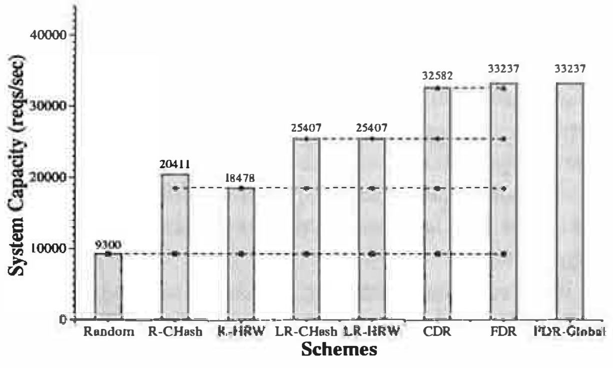


Figure 5: Capacity Comparison under Normal Load

5 Results

This section evaluates how the different strategies in Table 1 perform, both under normal conditions and under flash crowds or DDoS attacks. Network proximity and other factors that affect the performance of these strategies are also addressed.

5.1 Normal Workload

Before evaluating these strategies under flash crowds or other attack, we first measure their behavior under normal workloads. In these simulations, all clients generate traffic similar to normal users and gradually increase their request rates as discussed in Section 4.3. We compare aggregate system capacity and user-perceived latency under the different strategies, using the topology shown in Figure 4.

5.1.1 Optimal Static Replication

The static replication schemes (R-CHash, R-HRW, and their variants) use a configurable (but fixed) number of replicas, and this parameter's value influences their performance. Using a single replica per URL perfectly partitions the file set, but can lead to early failure of servers hosting popular URLs. Using as many replicas as available servers degenerates to the Random strategy. To determine an appropriate value, we varied this parameter between 2 and 64 replicas for R-CHash when there are 64 servers available. Increasing the number of replicas per URL initially helps to improve the system's throughput as the load gets more evenly distributed. Beyond a certain point, throughput starts decreasing due to the fact that each server is presented with a larger working set, causing more disk activity. In the 64-server case—the scenario we use throughout the rest of this section—10 server replicas for each URL achieves the optimal system capacity. For all of the remaining experiments, we use this value in the R-CHash and R-HRW schemes and their variants.

5.1.2 System Capacity

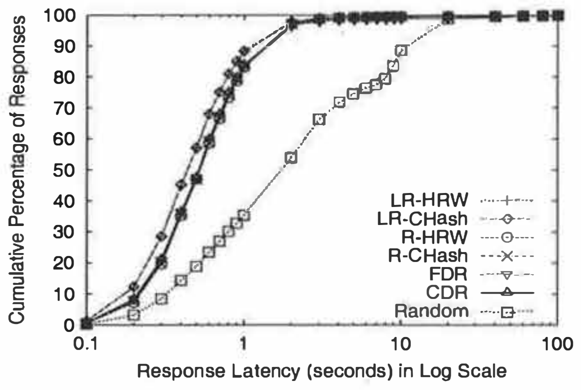
The maximum aggregate throughput of the various strategies with 64 servers are shown in Figure 5. Here we do not plot all the strategies and variants, but focus on those impacting throughput substantially. Random shows the lowest throughput at 9,300 req/s before overload. The static replication schemes, R-CHash and R-HRW, outperform Random by 119% and 99%, respectively. Our approximation of static schemes' best behaviors, LR-CHash and LR-HRW, yields 173% better capacity than Random. The dynamic replication schemes, CDR and FDR, show over 250% higher throughput than Random, or more than a 60% improvement over the static approaches and 28% over static schemes with fine-grained load control.

The difference between Random and the static approaches stems from the locality benefits of the hashing in the static schemes. By partitioning the working set, more documents are served from memory by the servers. Note, however, that absolute minimal replication can be detrimental, and in fact, the throughput for only two replicas in in Section 5.1.1 is actually lower than the throughput for Random. The difference in throughput between R-CHash and R-HRW is 10% in our simulation. However, this difference should not be over emphasized, because changes in the number of servers or workload can cause their relative ordering to change. Considering load helps the static schemes gain about 25% better throughput, but they still do not exceed the dynamic approaches.

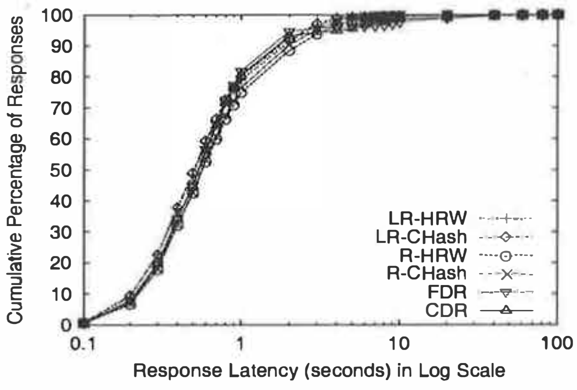
The performance difference between the static (including with load control) and dynamic schemes stems from the adjustment of the number of replicas for the documents. FDR also shows 2% better capacity than CDR.

Interestingly, the difference between our dynamic schemes (with only local knowledge) and the FDR-Global policy (with perfect global knowledge) is minimal. These results suggest that request distribution policies not only fare well with only local information, but that adding more global information may not gain much in system capacity.

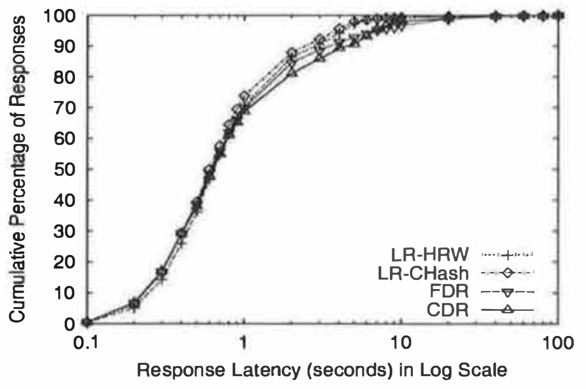
Examination of what ultimately causes overload in these systems reveals that, under normal load, the server's behavior is the factor that determines the performance limit of the system. None of the schemes suffers from saturated network links in these non-attack simulations. For Random, due to the large working set, the disk performance is the limit of the system, and before system failure, the disks exhibit almost 100% activity while the CPU remains largely idle. The R-CHash, R-HRW and LR-CHash and LR-HRW exhibit much lower disk utilization at comparable request rates; but by the time the system becomes overloaded, their bottleneck



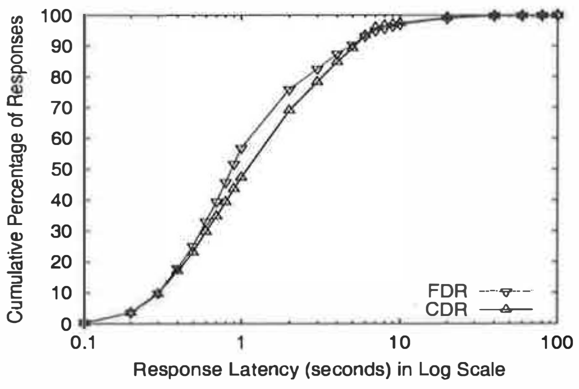
(a) Random's limit: 9,300 req/s



(b) R-HRW's limit: 18,478 req/s



(c) LR-HRW's limit: 25,407 req/s



(d) CDR's limit: 32,582 req/s

Figure 6: Response Latency Distribution under Normal Load

Utilization Scheme	CPU (%)		DISK (%)	
	Mean	Stddev	Mean	Stddev
Random	21.03	1.36	100.00	0.00
R-CHash	57.88	18.36	99.15	3.89
R-HRW	47.88	15.33	99.74	1.26
LR-CHash	59.48	18.85	97.83	12.51
LR-HRW	58.43	16.56	99.00	5.94
CDR	90.07	11.78	36.10	25.18
FDR	93.86	7.58	33.96	20.38
FDR-Global	91.93	11.81	17.60	15.43

Table 2: Server Resource Utilization at Overload

also becomes the disk and the CPU is roughly 50-60% utilized on average. In the CDR and FDR cases, at system overload, the average CPU is over 90% busy, while most of the disks are only 10-70% utilized. Table 2 summarizes resource utilization of different schemes before server failures (not at the same time point).

These results suggest that the CDR and FDR schemes are the best suited for technology trends, and can most benefit from upgrading server capacities. The throughput of our simulated machines is lower than what can be expected from state-of-the-art machines, but this decision to scale down resources was made to keep the simulation time manageable. *With faster simulated machines, we expect the gap between the dynamic schemes and the others to grow even larger.*

5.1.3 Response Latency

Along with system capacity, the other metric of interest is user-perceived latency, and we find that our schemes also perform well in this regard. To understand the latency behavior of these systems, we use the capacity measurements from Figure 5 and analyze the latency of all of the schemes whenever one category reaches its performance limit. For schemes with similar performance in the same category, we pick the lower limit for the analysis so that we can include numbers for the higher-performing scheme. In all cases, we present the cumulative distribution of all request latencies as well as some statistics

Req Rate	*9,300 req/s*				*18,478 req/s*				*25,407 req/s*				*32,582 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	3.95	1.78	11.32	6.99												
R-CHash	0.79	0.53	1.46	2.67	1.01	0.57	1.98	3.58								
R-HRW	0.81	0.53	1.49	2.83	1.07	0.57	2.28	3.22								
LR-CHash	0.68	0.44	1.17	2.50	0.87	0.51	1.82	2.74	1.19	0.60	2.47	3.79				
LR-HRW	0.68	0.44	1.18	2.50	0.90	0.51	1.89	3.13	1.27	0.64	2.84	3.76				
CDR	1.16	0.52	1.47	5.96	1.35	0.55	1.75	6.63	1.86	0.63	4.49	6.62	2.37	1.12	5.19	7.21
FDR	1.10	0.52	1.48	5.49	1.35	0.54	1.64	6.70	1.87	0.62	3.49	6.78	2.22	0.87	4.88	7.12
FDR-Global	0.78	0.50	1.42	2.88	0.97	0.54	1.58	5.69	1.11	0.56	1.86	5.70	1.35	0.66	2.35	6.29

Table 3: Response Latency of Different Strategies under Normal Load. μ — Mean, σ — Standard Deviation.

about the distribution.

Figure 6 plots the cumulative distribution of latencies at four request rates: the maximums for Random, R-HRW, LR-HRW, and CDR (the algorithm in each category with the smallest maximum throughput). The x-axis is in log scale and shows the time needed to complete requests. The y-axis shows what fraction of all requests finished in that time. The data in Table 3 gives mean, median, 90th percentile and standard deviation details of response latencies at our comparison points.

The response time improvement from exploiting locality is most clearly seen in Figure 6a. At Random's capacity, most responses complete under 4 seconds, but a few responses take longer than 40 seconds. In contrast, all other strategies have median times almost one-fourth that of Random, and even their 90th percentile results are less than Random's median. These results, coupled with the disk utilization information, suggest that most requests in the Random scheme are suffering from disk delays, and that the locality improvement techniques in the other schemes are a significant benefit.

The benefit of FDR over CDR is visible in Figure 6d, where the plot for FDR lies to the left of CDR. The statistics also show a much better median response time, in addition to better mean and 90th percentile numbers. FDR-Global has better numbers in all cases than CDR and FDR, due to its perfect knowledge of server load status.

An interesting observation is that when compared to the static schemes, dynamic schemes have worse mean times but comparable/better medians and 90th percentile results. We believe this behavior stems from the time required to serve the largest files. Since these files are less popular, the dynamic schemes replicate them less than the static schemes do. As a result, these files are served from a smaller set of servers, causing them to be served more slowly than if they were replicated more widely. We do not consider this behavior to be a significant drawback, and note that some research explicitly aims to achieve this effect [10, 11]. We will revisit large file issues in section 5.4.2.

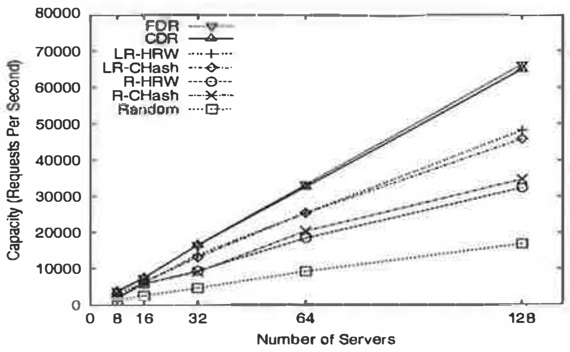


Figure 7: System Scalability under Normal Load

5.1.4 Scalability

Robustness not only comes from resilience with certain resources, but also from good scalability with increasing resources. We repeat similar experiments with different number of servers, from 8 to 128, to test how well these strategies scale. The number of server-side routers is not changed, but instead, more servers are attached to each server router as the total number of servers increases.

We plot system capacity against the number of servers in Figure 7. They all display near-linear scalability, implying all of them are reasonably good strategies when the system becomes larger. Note, for CDR and FDR with 128 servers, our original network provision is a little small. The bottleneck in that case is the link between the server router and backbone router, which is 622Mbps. In this scenario, each server router is handling 16 servers, giving each server on average only 39Mbps of traffic. At 600 reqs/s, even an average size of 10KB requires 48Mbps. Under this bandwidth setup, CDR and FDR yield similar system capacity as LR-CHash and LR-HRW, and all these 4 strategies saturate server-router-to-backbone links. To remedy this situation, we run simulations of 128 servers for all strategies with doubled bandwidth on both the router-to-backbone and backbone links. Performance numbers of 128 servers under these faster links are plotted in the graph instead. This problem can also be solved by placing fewer servers behind each pipe and instead spreading them across more locations.

5.2 Behavior Under Flash Crowds

Having established that our new algorithms perform well under normal workloads, we now evaluate how they behave when the system is under a flash crowd or DDoS attack. To simulate a flash crowd, we randomly select 25% of the 1,000 clients to be *intensive* requesters, where each of these requesters repeatedly issues requests from a small set of pre-selected URLs with an average size of about 6KB.

5.2.1 System Capacity

Figure 8 depicts the system capacity of 64 servers under a flash crowd with a set of 10 URLs. In general, it exhibits similar trends as the no-attack case shown in Figure 5. Importantly, the CDR and FDR schemes still yield the best throughput, making them most robust to flash crowds or attacks. Two additional points deserve more attention.

First, FDR now has a similar capacity with CDR, but still is more desirable as it provides noticeably better latency, as we will see later. FDR's benefit over R-CHash and R-HRW has grown to 91% from 60% and still outperforms LR-CHash and LR-HRW by 22%.

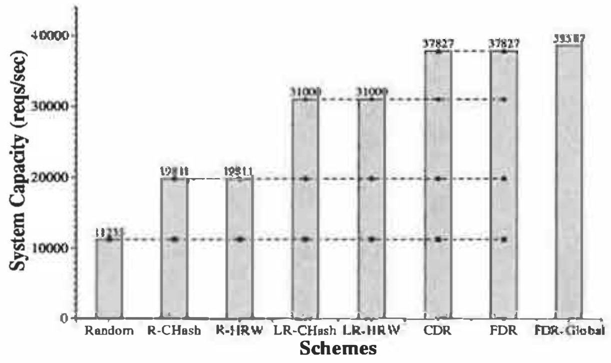


Figure 8: Capacity Comparison Under Flash Crowds

Second, the absolute throughput numbers tend to be larger than the no-attack case, because the workload is also different. Here, 25% of the traffic is now concentrated on 10 URLs, and these attack URLs are relatively small, with an average size of 6KB. Therefore, relative difference among different strategies within each scenario yields more useful information than simply comparing performance numbers across these two scenarios.

5.2.2 Response Latency

The cumulative distribution of response latencies for all seven algorithms under attack are shown in Figure 9. Also, the statistics for all seven algorithms and FDR-Global are given in Table 4. As seen from the figure and table, R-CHash, R-HRW, LR-CHash, LR-HRW CDR and FDR still have far better latency than Random, and static schemes are a little better than CDR and FDR at Random, R-HRW's and LR-HRW's failure points; and LR-CHash and LR-HRW yields slightly better latency than R-CHash and R-HRW.

As we explained earlier, CDR and FDR adjust the server replica set in response to request volume. The number of replicas that serve attack URLs increases as the attack ramps up, which may adversely affect serving non-attack URLs. However, the differences in the mean, median, and 90-percentile are not large, and all are probably acceptable to users. The small price paid in response time for CDR and FDR brings us higher system capacity, and thus, stronger resilience to various loads.

5.2.3 Scalability

We also repeat the scalability test under flash crowd or attack, where 250 clients are *intensive* requesters that repeatedly request 10 URLs. As shown in Figure 10, all strategies scale linearly with the number of servers. Again, in the 128-server cases, we use doubled bandwidth on the router-to-backbone and backbone links.

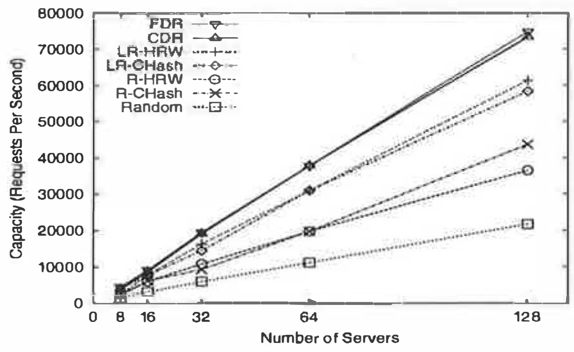
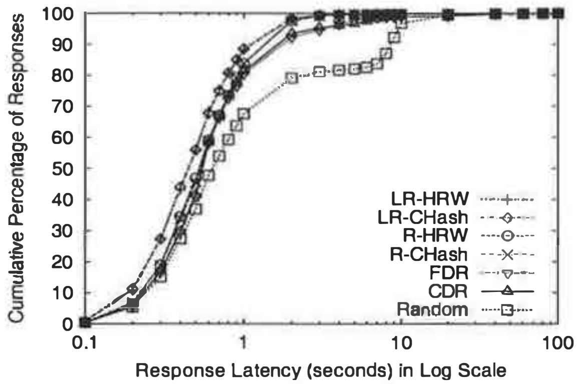


Figure 10: System Scalability under Flash Crowds

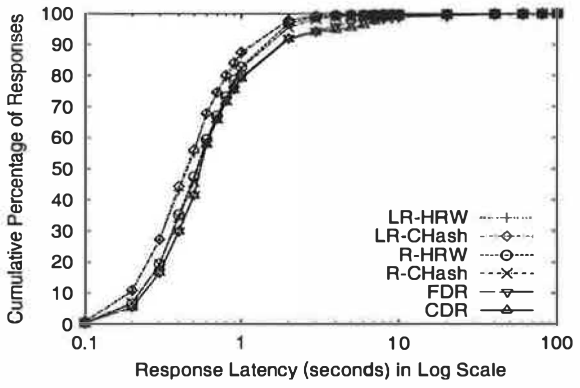
5.2.4 Various Flash Crowds

Throughout our simulations, we have seen that a different number of *intensive* requesters, and a different number of hot or attacked URLs, have an impact on system performance. To further investigate this issue, we carry out a series of simulations by varying both the number of intensive requesters and the number of hot URLs. Since it is impractical to exhaust all possible combinations, we choose two classes of flash crowds. One class has a single hot URL of size 1KB. This represents a small home page of a website. The other class has 10 hot URLs averaging 6KB, as before. In both cases, we vary the percentage of the 1000 clients that are intensive requesters from 10% to 80%. The results of these two experiments with 32 servers are shown in Figures 11 and 12, respectively.

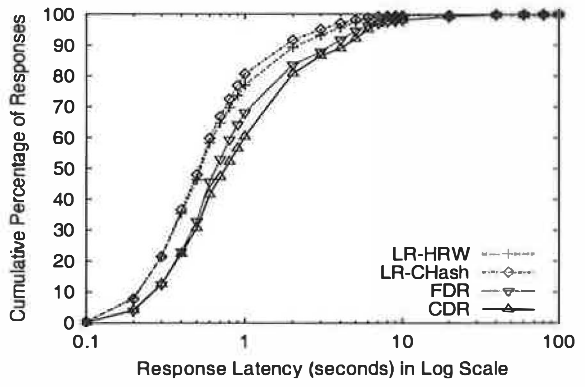
In the first experiment, as the portion of *intensive* requesters increases, more traffic is concentrated on this one URL, and the request load becomes more unbalanced. Random, CDR and FDR adapt to this change



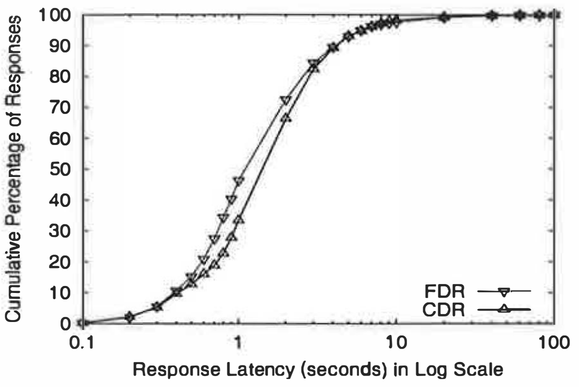
(a) Random's limit: 11,235 req/s



(b) R-HRW's limit: 19,811 req/s



(c) LR-HRW's limit: 31,000 req/s



(d) CDR's limit: 37,827 req/s

Figure 9: Response Latency Distribution under Flash Crowds

well and yield increasing throughput. This benefit comes from their ability to spread load across more servers. However, CDR and FDR behave better than Random because they not only adjust the server replica set on demand, but also maintain server locality for less popular URLs. In contrast, R-HRW, R-CHash, LR-HRW and LR-CHash suffer with more intensive requesters or attackers, since their fixed number of replicas for each URL cannot handle the high volume of requests for one URL. In the 10-URL case, the change in system capacity looks similar to the 1-URL case, except that due to more URLs being intensively requested or attacked, FDR, CDR and Random cannot sustain the same high throughput. We continue to investigate the effects of more attack URLs and other strategies.

Another possible DDoS attack scenario is to randomly select a wide range of URLs. In the case that these URLs are valid, the dynamic schemes "degenerate" into one server for each URL. This is the desirable behavior for this attack as it increases the cache hit rates for all the servers. In the event that the URLs are invalid, and the servers are actually reverse proxies (as is typically the case in a CDN), then these invalid URLs are forwarded to the server-of-origin, effectively overloading it. Servers must address this possibility by throttling the number of URL-misses they forward.

To summarize, under flash crowds or attacks, CDR and FDR sustain very high request volumes, making overloading the whole system significantly harder and thereby greatly improving the CDN system's overall robustness.

5.3 Proximity

The previous experiments focus on system capacity under different loads. We now compare the strategies that factor network closeness into server selection—Static (NPR-CHash), Static+Load (NPLR-CHash), and Dynamic (NP-FDR)—with their counterparts that ignore proximity. We test the 64-server cases in the same scenarios as in Section 5.1 and 5.2.

Req Rate	*11,235 req/s*				*19,811 req/s*				*31,000 req/s*				*37,827 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	2.37	0.64	8.57	5.29												
R-CHash	0.73	0.53	1.45	2.10	0.81	0.53	1.57	2.59								
R-HRW	0.73	0.52	1.45	2.11	0.76	0.52	1.51	2.51								
LR-CHash	0.62	0.45	1.15	1.70	0.67	0.45	1.23	2.42	0.96	0.52	1.86	3.55				
LR-HRW	0.63	0.45	1.18	1.80	0.67	0.46	1.26	2.65	1.07	0.53	2.19	3.52				
CDR	1.19	0.55	1.72	5.40	1.25	0.55	1.86	5.51	1.80	0.76	4.35	6.08	2.29	1.50	4.20	6.41
FDR	1.22	0.55	1.81	5.71	1.18	0.55	1.83	5.27	1.64	0.66	3.57	5.95	2.18	1.14	4.15	6.63
FDR-Global	0.91	0.55	1.66	4.09	0.90	0.53	1.60	4.59	0.98	0.54	1.74	5.08	1.20	0.56	1.99	5.53

Table 4: Response Latency of Different Strategies under Flash Crowds. μ — Mean, σ — Standard Deviation.

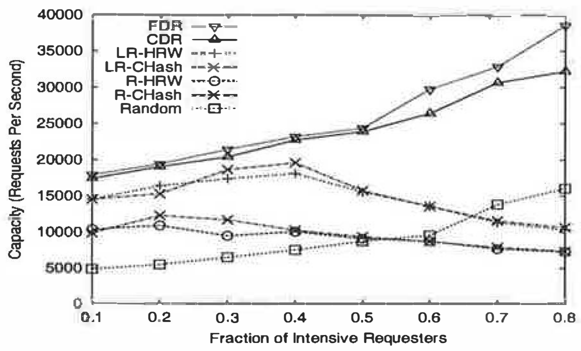


Figure 11: 1 Hot URL, 32 Servers, 1000 Clients

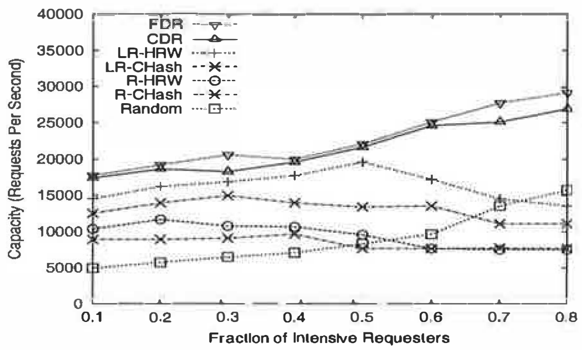


Figure 12: 10 Hot URL, 32 Servers, 1000 Clients

	System Capacity (reqs/sec)		
Category	Scheme	Normal	Flash Crowds
Static	*NPR-CHash*	14409	14409
	R-CHash	20411	19811
Static +Load	*NPLR-CHash*	24173	30090
	LR-CHash	25407	31000
Dynamic	*NP-FDR*	31000	34933
	FDR	33237	37827

Table 5: Proximity's Impact on Capacity

Table 5 shows the capacity numbers of these strategies under both normal load and flash crowds of 250 intensive requesters with 10 hot URLs. As we can see, adding network proximity into server selection slightly decreases systems capacity in the case of NPLR-CHash and NP-FDR. However, the throughput drop of NPR-CHash compared with R-CHash is considerably large. Part of reason is that in LR-CHash and FDR, server load information already conveys the distance of a server. However, in the R-CHash case, the redirector randomly choosing among all replicas causes the load to be evenly distributed, while NPR-CHash puts more burden on closer servers, resulting in unbalanced server load.

We further investigate the impact of network proximity on response latency. In Table 6 and 7, we show the latency statistics under both normal load and flash crowds. As before, we choose to show numbers at the capacity limits of Random, NPR-CHash, NPLR-CHash and NP-FDR. We can see that when servers are not loaded, all schemes with network proximity taken into consideration—NPR-CHash, NPLR-CHash and NP-FDR—yield better latency. When these schemes reach their limit, NPR-CHash and NP-FDR still demonstrate significant latency advantage over R-CHash and FDR, respectively.

Interestingly, NPLR-CHash underperforms LR-CHash in response latency at its limit of 24,173 req/s and 30,090 req/s. NPLR-CHash is basically LR-CHash using effective load. When all the servers are not loaded, it redirects more requests to nearby servers, thus shortening the response time. However, as the load increases, in order for a remote server to get a share of load, a local server has to be much more overloaded than the remote one, inversely proportional to their distance ratio. Unlike NP-FDR, there is no load threshold control in NPLR-CHash, so it is possible that some close servers get significantly more requests, resulting in slow processing and longer responses. In a summary, considering proximity may benefit latency, but it can also impact capacity. NP-FDR, however, achieves a good balance of both.

5.4 Other Factors

5.4.1 Heterogeneity

To determine the impact of network heterogeneity on our schemes, we explore the impact of non-uniform server network bandwidth. In our original setup, all first-mile links from the server have bandwidths of 100Mbps. We now randomly select some of the servers and re-

Req Rate	*9,300 req/s*				*14,409 req/s*				*24,173 req/s*				*31,000 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	3.95	1.78	11.32	6.99												
NPR-CHash	0.66	0.42	1.21	2.20	0.76	0.44	1.51	2.30								
R-CHash	0.79	0.53	1.46	2.67	0.82	0.56	1.63	2.50								
NPLR-CHash	0.57	0.36	0.93	2.00	0.68	0.39	1.33	2.34	1.34	0.55	2.63	4.73				
LR-CHash	0.68	0.44	1.17	2.50	0.71	0.48	1.43	2.19	1.04	0.50	1.95	3.44				
NP-FDR	0.70	0.50	1.42	1.63	0.67	0.49	1.33	1.56	0.80	0.49	1.55	2.82	1.08	0.53	1.96	3.54
FDR	1.10	0.52	1.48	5.49	1.25	0.54	1.71	5.87	1.60	0.57	2.10	6.84	1.88	0.59	3.72	7.25

Table 6: Proximity's Impact on Response Latency under Normal Load. μ — Mean, σ — Standard Deviation.

Req Rate	*11,235 req/s*				*14,409 req/s*				*30,090 req/s*				*34,933 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
Random	2.37	0.64	8.57	5.29												
NPR-CHash	0.61	0.42	1.15	1.76	0.63	0.41	1.08	2.34								
R-CHash	0.73	0.53	1.45	2.10	0.73	0.52	1.38	2.50								
NPLR-CHash	0.53	0.36	0.90	1.75	0.55	0.35	0.91	2.29	1.29	0.61	2.65	3.94				
LR-CHash	0.62	0.45	1.15	1.70	0.64	0.44	1.13	2.56	0.90	0.49	1.73	3.44				
NP-FDR	0.70	0.50	1.45	1.68	0.66	0.45	1.34	1.63	0.81	0.47	1.64	2.55	0.99	0.51	1.92	3.26
FDR	1.22	0.55	1.81	5.71	1.07	0.54	1.67	5.47	1.60	0.66	3.49	5.90	1.84	0.78	4.15	6.31

Table 7: Proximity's Impact on Response Latency under Flash Crowds. μ — Mean, σ — Standard Deviation.

duce their link bandwidth by an order of magnitude, to 10Mbps. We want to test how different strategies respond to this heterogeneous environment. We pick representative schemes from each category: Random, R-CHash, LR-CHash and FDR and stress them under both normal load and flash crowd similar to network proximity case. Table 8 summarizes our findings on system capacities with 64 servers.

Redirection	Portion of Slower Links					
	Normal Load			Flash Crowds		
Schemes	0%	10%	30%	0%	10%	30%
Random	9300	8010	8010	11235	8449	8449
R-CHash	20411	7471	7471	19811	7110	7110
LR-CHash	25407	23697	19421	31000	26703	22547
FDR	33237	31000	25407	37827	34933	29496

Table 8: Capacity (reqs/sec) with Heterogeneous Server Bandwidth,

From the table we can see, under both normal load and flash crowds, Random and R-CHash are hurt badly because they are load oblivious and keep assigning requests to servers with slower links thereby overload them early. In contrast, LR-CHash and FDR only suffer slight performance downgrade. However, FDR still maintains advantage over LR-CHash, due to its dynamic expanding of server set for hot URLs.

5.4.2 Large File Effects

As we discussed at the end of section 5.1.3, the worse mean response times of dynamic schemes come from serving large files with a small server set. Our first attempt to remedy this situation is to handle the largest files specially. Analysis of our request trace indicates that 99% of the files are smaller than 530KB, so we use this value as a threshold to trigger special large file treatment. For these large files, there are two simple ways to redirect requests for them. One is to redirect these requests to a random server, which we call T-R (tail-random). The other is to redirect these requests to a least loaded member in a server set of fixed size (larger than one), which we call T-S (tail-static). Both of these approaches enlarge the server set serving large files. We repeat experiments of 64 server cases in Section 5.1 and 5.2 using these two new approaches, where T-S employs a 10-replica server set for large files in the distribution tail. Handling the tail specially yields slightly better capacity than standard CDR or FDR, but the latency improves significantly. Table 9 summarizes latency results under normal load. As we can see, the T-R and T-S versions of CDR and FDR usually generate better latency numbers than LR-CHash and LR-HRW. Results under flash crowds are similar. This confirms our assertion about large file effects.

6 Related Work and Discussion

Cluster Schemes: Approaches for request distribution in clusters [8, 12, 17] generally use a switch/router through which all requests for the cluster pass. As a result, they can use various forms of feedback and load information from servers in the cluster to improve system performance. In these environments, the delay between the redirector and the servers is minimal, so they can have tighter coordination [2] than in schemes like ours, which are developed for wide-area environments. We do, however, adapt the fine-grained server set accounting from the LARD/R approach [25] for our Fine Dynamic Replication approach.

Req Rate	*9,300 req/s*				*18,478 req/s*				*25,407 req/s*				*32,582 req/s*			
Latency	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ	μ	50%	90%	σ
LR-CHash	0.68	0.44	1.17	2.50	0.87	0.51	1.82	2.74	1.19	0.60	2.47	3.79				
LR-HRW	0.68	0.44	1.18	2.50	0.90	0.51	1.89	3.13	1.27	0.64	2.84	3.76				
CDR	1.16	0.52	1.47	5.96	1.35	0.55	1.75	6.63	1.86	0.63	4.49	6.62	2.37	1.12	5.19	7.21
CDR-T-R	0.78	0.52	1.43	2.77	0.76	0.52	1.40	2.80	1.05	0.57	1.90	3.06	1.58	0.94	3.01	3.55
CDR-T-S	0.74	0.52	1.43	2.17	0.72	0.52	1.38	2.44	1.01	0.56	1.93	2.96	1.53	0.68	3.69	4.18
FDR	1.10	0.52	1.48	5.49	1.35	0.54	1.64	6.70	1.87	0.62	3.49	6.78	2.22	0.87	4.88	7.12
FDR-T-R	0.78	0.52	1.43	2.77	0.75	0.52	1.40	2.82	1.01	0.57	1.87	2.98	1.39	0.77	2.82	3.68
FDR-T-S	0.74	0.52	1.43	2.17	0.72	0.52	1.37	2.55	0.98	0.56	1.84	2.95	1.41	0.63	2.88	3.88

Table 9: Response Latency with Special Large File Handling, Normal Load. μ — Mean, σ — Standard Deviation.

Distributed Servers: In the case of geographically distributed caches and servers, DNS-based systems can be used to obliviously spread load among a set of servers, as in the case of round-robin DNS [4], or it can be used to take advantage of geographically dispersed server replicas [6]. More active approaches [9, 14, 16] attempt to use load/latency information to improve overall performance. We are primarily focused on balancing load, locality and latency, meanwhile, we also demonstrate a feasible way to incorporate network proximity into server selection explicitly.

Web Caches: We have discussed proxy caches as one deployment vehicle for redirectors, and these platforms are also used in other content distribution schemes. The simplest approach, the static cache hierarchy [7], performs well in small environments but fails to scale to much larger populations [32]. Other schemes involve overlapping meshes [33] or networks of caches in a content distribution network [19], presumably including commercial CDNs such as Akamai.

DDoS Detection and Protection: DDoS attacks have become an increasingly serious problem on the Internet [22]. Researchers have recently developed techniques to identify the source of attacks using various traceback techniques, such as probabilistic packet marking [28] and SPIE [29]. These approaches are effective in detecting and confining attack traffic. With their success in deterring spoofing and suspicious traffic, attackers have to use more disguised attacks, for example by taking control of large number of slave hosts and instructing them to attack victims with legitimate requests. Our new redirection strategy is effective in providing protection against exactly such difficult-to-detect attacks.

Peer-to-Peer Networks: Peer-to-peer systems provide an alternative infrastructure for content distribution. Typical peer-to-peer systems involve a large number of participants acting as both clients and servers, and they have the responsibility of forwarding traffic on behalf of others. Given their very large scale and massive resources, peer-to-peer networks could provide a potential robust means of information dissemination or exchange. Many peer-to-peer systems, such as CAN [26], Chord [30], and Pastry [27] have been proposed and they can serve as a substrate to build other services. Most of these peer-to-peer networks use a distributed hash-based scheme to combine object location and request routing and are designed for extreme scalability up to hundreds of thousands of nodes and beyond. We also use a hash-based approach, but we are dealing one to two orders of magnitude fewer servers than the peers in these systems, and we expect relatively stable servers. As a result, much of the effort that peer-to-peer networks spend in discovery and membership issues is not needed for our work. Also, we require fewer intermediaries between the client and server, which may translate to lower latency and less aggregate network traffic.

7 Conclusions

This paper demonstrates that improved request redirection strategies can effectively improve CDN robustness by balancing locality, load and proximity. Detailed end-to-end simulations show that even when redirectors have imperfect information about server load, algorithms that dynamically adjust the number of servers selected for a given object, such as FDR, allow the system to support a 60-91% greater load than best published CDN systems. Moreover, this gain in capacity does not come at the expense of response time, which is essentially the same both when the system is under flash crowds and when operating under normal conditions.

These results demonstrate that the proposed algorithm results in a system with significantly greater capacity than published CDNs, which should improve the system's ability to handle legitimate flash crowds. The results also suggest a new strategy in defending against DDoS attacks: each server added to the system multiplicatively increases the number of resources an attacker must marshal in order to have a noticeable affect on the system.

Although we believe this paper identifies important trends, much work remains to be done. We have conducted the largest detailed simulations as current simulation environment allows. We also find that approximate load information works well. We expect our new algorithms scale to very large systems with thousands of servers, but it requires a lot more resources and time to evaluate. We would like to run simulations at an even larger scale, with faster, more powerful simulated servers. We would also like to experiment with more topologies such as those generated by power-law based topology generators, use more traces, real or synthetic (such as SPECweb99). Finally, we plan to deploy our new algorithms on a testbed and explore other implementation issues.

Acknowledgments

This research is supported in part by Compaq and DARPA contract F30602–00–2–0561. We thank our shepherd, David Wetherall, for his guidance and helpful inputs. We also thank our anonymous reviewers for their valuable comments on improving this paper.

8 REFERENCES

[1] Akamai. Akamai content delivery network. http://www.akamai.com.

[2] D. Andresen, T. Yang, V. Holmedahl, and O. Ibarra. Sweb: Towards a scalable world wide web server on multicomputers, 1996.

[3] A. Barbir, B. Cain, F. Douglis, M. Green, M. Hofmann, R. Nair, D. Potter, and O. Spatscheck. Known CN Request-Routing Mechanisms, Feb. 2002. Work in Progress, draft-ietf-cdi-known-request-routing-00.txt.

[4] T. Brisco. DNS support for load balancing. Request for Comments 1794, Rutgers University, New Brunswick, New Jersey, Apr. 1995.

[5] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies an d Systems (USITS)*, Monterey, CA, Dec. 1997.

[6] V. Cardellini, M. Colajanni, and P. Yu. Geographic load balancing for scalable distributed web systems. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Aug. 2000.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[8] J. Cohen, N. Phadnis, V. Valloppillil, and K. W. Ross. Cache array routing protocol v1.1. http://ds1.internic.net/internet-drafts/draft-vinod-carp-v1-01.txt, September 1997.

[9] M. Colajanni, P. S. Yu, and V. Cardellini. Dynamic load balancing in geographically distributed heterogeneous web servers. In *International Conference on Distributed Computing Systems*, pages 295–302, 1998.

[10] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[11] M. Crovella, M. Harchol-Balter, and C. D. Murta. Task assignment in a distributed system: Improving performance by unbalancing load (extended abstract). In *Measurement and Modeling of Computer Systems*, pages 268–269, 1998.

[12] O. Damani, P. Y. Chung, Y. Huang, C. M. R. Kintala, and Y. M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. In *Proceedings of the Sixth International World-Wide Web Conference*, 1997.

[13] Digital Island. http://www.digitalisland.com.

[14] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *INFOCOM (2)*, pages 783–791, 1998.

[15] L. Garber. Technology news: Denial-of-service attacks rip the Internet. *Computer*, 33(4):12–17, Apr. 2000.

[16] J. D. Guyton and M. F. Schwartz. Locating nearby copies of replicated internet servers. In *SIGCOMM*, pages 288–298, 1995.

[17] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.

[18] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek. The measured performance of content distribution networks. In *Proceedings of The 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[19] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceedings of the Eighth International World-Wide Web Conference*, 1999.

[20] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654-663, 1997.

[21] Mirror Image. http://www.mirror-image.com.

[22] D. Moore, G. Voelker, and S. Savage. Inferring internet denial of service activity. In *Proceedings of 2001 USENIX Security Symposium*, Aug. 2001.

[23] NS. (Network Simulator). http://www.isi.edu/nsnam/ns/.

[24] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference*, June 1999.

[25] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-aware request distribution in cluster-based network servers. In *Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.

[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM'01*, Aug. 2001.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[28] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, Aug. 2000.

[29] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based ip traceback. In *Proceedings of ACM SIGCOMM'01*, Aug. 2001.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM'01*, Aug. 2001.

[31] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, Feb. 1998.

[32] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles*, pages 16–31, 1999.

[33] L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching. In *Proceedings of the 1997 NLANR Web Cache Workshop*, June 1997.

The Design and Implementation of Zap: A System for Migrating Computing Environments

Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh
Department of Computer Science
Columbia University
{sto8, dinesh, gongsu, nieh}@cs.columbia.edu

Abstract

We have created Zap, a novel system for transparent migration of legacy and networked applications. Zap provides a thin virtualization layer on top of the operating system that introduces pods, which are groups of processes that are provided a consistent, virtualized view of the system. This decouples processes in pods from dependencies to the host operating system and other processes on the system. By integrating Zap virtualization with a checkpoint-restart mechanism, Zap can migrate a pod of processes as a unit among machines running independent operating systems without leaving behind any residual state after migration. We have implemented a Zap prototype in Linux that supports transparent migration of unmodified applications without any kernel modifications. We demonstrate that our Linux Zap prototype can provide general-purpose process migration functionality with low overhead. Our experimental results for migrating pods used for running a standard user's X windows desktop computing environment and for running an Apache web server show that these kinds of pods can be migrated with subsecond checkpoint and restart latencies.

1 Introduction

Process migration is the ability to transfer a process from one machine to another. It is a useful facility in distributed computing environments, especially as computing devices become more pervasive and Internet access becomes more ubiquitous. The potential benefits of process migration, among others, are fault resilience by migrating processes off of faulty hosts, data access locality by migrating processes closer to the data, better system response time by migrating processes closer to users, dynamic load balancing by migrating processes to less loaded hosts, and improved service availability and administration by migrating processes before host maintenance so that applications can continue to run with minimal downtime.

Although process migration provides substantial potential benefits and many approaches have been considered [24], achieving process migration functionality has been difficult in practice. Toward this end, there are four important goals that need to be met. First, given the large number of widely used legacy applications, applications should be able to migrate and continue to operate correctly without modification, without requiring that they be written using uncommon languages or toolkits, and without restricting their use of common operating system services. For example, networked applications should be able to maintain their network connections even after being migrated. Second, migration should leverage the large existing installed base of commodity operating systems. It should not necessitate use of new operating systems or substantial modifications to existing ones. Third, migration should maintain the independence of independent machines. It should avoid creating residual dependencies that limit the utility of process migration by requiring machines where a process was previously executed to continue to service a process even after it has migrated to another machine. Fourth, migration should be fast and efficient. Overhead should be small for normal execution and migration.

To overcome limitations in previous approaches to general-purpose process migration, we have created Zap. Zap provides a thin virtualization layer on top of the operating system that introduces a *PrOcess Domain* (pod) abstraction. A pod provides a group of processes with a private namespace that presents the process group with the same virtualized view of the system. This virtualized view associates virtual identifiers with operating system resources such as process identifiers and network addresses. This decouples processes in a pod from dependencies on the host operating system and from other processes in the system.

Zap virtualization is integrated with a checkpoint-restart mechanism that enables processes within a pod to be migrated as a unit to another machine. Since pods are independent and self-contained they can be migrated freely without leaving behind any residual state after migration, even when migrating network applications while preserving their network connections. Zap can therefore allow applications to continue executing after migration even if the machine on which they previously executed is no longer available. In using a checkpoint-restart approach, Zap not only supports process migration, but also allows processes to be suspended to secondary storage and transparently resumed at a later time.

Going beyond simple migration, this functionality can be useful in many ways, including fast creation of user sessions and simpler, more dynamic system configuration.

Zap is designed to support migration of unmodified legacy applications while minimizing changes to existing operating systems. This is done by leveraging loadable kernel module functionality in commodity operating systems that allows Zap to intercept system calls as needed for virtualization and save and restore kernel state as needed for migration. Zap's compatibility with existing applications and operating systems makes it simple to deploy and use. We have implemented a Zap prototype as a loadable kernel module in Linux that supports transparent migration, without any kernel modifications, among separate machines running independent Linux operating systems; it does not require a single-system image across all machines. Our experimental results on our Linux Zap prototype demonstrate that it can provide general-purpose process migration functionality with low overhead.

This paper focuses on the design and implementation of the Zap virtualization and migration mechanisms. Section 2 describes related work. Section 3 describes the pod abstraction provided by Zap. Section 4 describes the architecture of Zap and the mechanisms that support the pod abstraction. Section 5 presents an overview of our implementation of Zap in Linux. Section 6 presents experimental results evaluate the overhead associated with Zap virtualization and migration mechanisms and demonstrate the utility of Zap for migrating legacy and network applications. Finally, we present concluding remarks and directions for future work.

2 Related Work

Many research operating systems have been developed that implemented process migration mechanisms, with a focus on using migration for load balancing. These systems include Accent [31], Amoeba [25], Charlotte [6], Chorus [33], MOSIX [7], Sprite [12], and V [11]. These operating systems provided a single system image across a cluster of machines and providing migration throughout the cluster through careful kernel design to provide a global namespace and location-transparent execution. Process state such as IPC, open files, and system calls in some cases are typically handled by forwarding requests to a home node on which the process originated. If the home node fails, migrated processes running on other nodes may fail as well. Although providing a single cluster operating system can simplify system management, these kinds of systems require new operating systems or substantial changes to existing ones, which have limited their deployment. Furthermore, these approaches do not work in the context of increasingly common clusters of independent machines, each with its own operating system.

Several systems have been developed to support process migration at the user-level and can be run on unmodified commercial operating systems. These systems include Condor [22], CoCheck [29], libckpt [28], and MPVM [10]. These systems are primarily intended for executing long-running applications on a cluster of machines. However, because there is no kernel support for process migration, these systems require processes to be well-behaved in order to migrate, which means that such processes cannot use common operating system services such as inter-process communication. This severely limits the kind of applications that can be used with such systems.

Several systems have been developed that provide migration using object-based approaches. These systems include Abacus [5], Emerald [19], Globus [13], Legion [14], and Rover [18]. These systems are designed as programming languages or middleware toolkits that typically require explicit programmer control to utilize migration. By operating at a higher-level of abstraction, these systems can reduce the amount of state that needs to be recorded and moved to migrate an application. However, these systems require applications to be re-written using new programming language environments. As a result, they cannot migrate legacy applications.

Virtualization at the operating system level has been proposed as a mechanism for supporting process migration. Zap virtualization was inspired in part by capsules [36], an abstraction that provided a private namespace to a group of processes that can be migrated as a unit. However, capsules did not support migration of networked applications while preserving their open network connections. Unlike Zap, implementing capsules required extensive operating system changes. Whereas the capsule approach restructured the operating system to achieve its goals, Zap seeks to be compatible with existing operating systems by virtualizing the operating system interface while minimizing changes to the operating system. Operating system virtualization has also been explored in vOS [9] to provide process migration for simple, non-networked Windows applications.

Virtual machine monitors (VMMs) can also be used as a mechanism for process migration [21, 35]. Virtual machine monitors such as VMware [1] virtualize at the hardware level to encapsulate an entire operating system environment such that it can be suspended and resumed. The operating system environment can be migrated from one machine to another assuming sufficient similarities in those system architectures. By leveraging VMware, even Microsoft Windows applications can be migrated without operating system or application changes, though the problem of migrating networked applications with open connections has not yet been addressed. Capsules have been recently applied in this context [35]. Because

VMMs operate below the operating system, they cannot take advantage of mechanisms specific to a given operating system to reduce the cost of migration and are limited to migrating an entire machine as opposed to a few processes. Furthermore, all applications to be migrated must be run using a VMM. Section 6 shows that the resulting cost of using VMMs can be substantially higher migration and runtime overhead.

Previous approaches to process migration do not effectively support networked applications. However, a variety of other approaches have been proposed to provide mobility for network communications. These approaches can be loosely categorized as network layer solutions [8, 17, 26], transport layer solutions [38], proxy-based solutions [23], and socket library wrapper solutions [30, 41]. Network layer solutions do not provide mobility for individual end-to-end transport connections; they only allow mobility at the level of an entire host. The proposed transport layer solution requires changes in the transport protocol and therefore is difficult to deploy. Existing proxy-based solutions are usually tied to a specific transport protocol (e.g., TCP) and their "connection switching and splicing" function incurs high overhead. Socket library wrapper solutions duplicate many transport protocol functions that result in high overhead.

3 The Pod Abstraction

The goal of Zap is to support migratable computing environments in the context of today's networked computing infrastructure. In this infrastructure, network file servers are typically used to store applications and user data. These servers are then accessible to personal computers or compute servers, where application processing takes place. A key characteristic of this computing environment is that the compute machines typically run completely independently of one another, each running its own independent operating system. Zap seeks to enable users to continue to use such a computing infrastructure as they normally do, but with the added feature of being able to have their computing sessions migrate across machines.

To support transparent process migration from one independent machine to another, Zap must address three key requirements regarding resource consistency, resource conflicts, and resource dependencies. The first requirement is the need to preserve resource naming consistency. An operating system contains numerous identifiers for its resources, including process IDs (PIDs), file names, and socket ports. Since neither the operating system nor the typical application were designed to support process migration, they both assume that these identifiers will remain constant throughout the life of the process. It is not unusual for a process to take note of an identifier given to it and store it in memory or on a file. In migrating a process from one machine to another, it is important to maintain consistent names for these resource identifiers to ensure that the process continues to function correctly.

The second requirement is the need to avoid potential resource naming conflicts in the presence of migrating processes. Since operating system resource identifiers are unique only at the system level, it is a trivial task for an operating system to produce unique identifiers merely by examining its current state and picking a candidate identifier that is not in use. The problem that can arise when a process is migrated to a new host system is that its process identifier could already be in use at the new host system. For example, a conflict will arise if a process with PID 20 is migrated into a system that already contains a process with PID 20. Besides aborting the migration, there are only two approaches at this point: (1) violate resource naming consistency and change the PID, which can cause common applications which have already stored their previous PIDs to fail because PID value has changed unexpectedly, or (2) wait and try again later after the PID resource becomes free, which can result in the migration never being able to occur since there is no way to know how long it will be before the resource becomes free.

The third requirement is the need to avoid creating dependencies among components of the system that cannot be easily severed when a process is migrated. For example, a process could attempt to share an area of memory with an operating system component such as another process, making it impossible to migrate the particular process unless: (1) the other component sharing memory is also migrated simultaneously, or (2) a proxy is left behind on the original host so that any updates of this particular shared memory area will be relayed through a network connection. The first approach may require larger and larger portions of the operating system to migrate at once. As the number of cross dependencies grows, the number of independently migratable processes decreases. The second approach is also unfavorable because application developers assume that accessing a shared memory area will be fast compared to network speeds. In addition, reliance on the original host machine is never removed, so that some of the potential advantages of process migration, such as freeing up a machine for maintenance, are lost.

To address these three requirements, Zap introduces a *pod* (PrOcess Domain) abstraction, which provides a collection of processes with a host-independent virtualized view of the operating system. Pods are self-contained units that can be suspended to secondary storage, migrated to another machine, and transparently resumed. A pod can contain any number of processes. For example, a pod can encapsulate all of the processes corresponding to a user's computing session. The abstraction provides the same application interface as the underlying operating system so that legacy applications can execute in the context of a pod without any modification. Processes within a pod can make use of all available operating system services, just like processes executing in a traditional operating system environment.

The main difference between a pod and a traditional operating system environment is that each pod has its own private, virtual namespace. The idea of a private, virtual namespace is surprisingly simple but has significant implications for supporting migratable computing environments. The pod namespace provides two key characteristics that facilitate the independence and mobility of pods.

First, the namespace provides consistent, virtual resource names in place of host-dependent resource names such as PIDs. Names within a pod are trivially assigned in a unique manner in the same way that traditional operating systems assign names, but such names are localized to the pod. Since the namespace is private to a given pod, there are no resource naming conflicts for processes in different pods. There is no need for the pod namespace to change when the pod is migrated, which allows pods to ensure that identifiers remain constant throughout the life of the process, as required by legacy applications that use such identifiers. Because pod namespaces are private, allowing a process to move from one pod to another could result in naming conflicts. As a result, processes are created inside of a pod and spend their entire lifetimes in the context of that pod; they are not allowed to leave one pod and join another.

Consider the following simple example of how the pod namespace aids in supporting process migration. The following excerpt of a C program is not atypical of a multi-threaded application:

```
int iChildPID;

if (iChildPID=fork()) {
  /* parent does some work */
  waitpid(iChildPID);/* Wait for the child
                         process to exit */
} else {
  /* child does some work */
  exit(0);
}
```

Migrating this program from a system A to a system B is trivial in the context of pods. If the child had a virtual PID of 172 in a pod on system A, it will still have the same PID in the same pod after it is migrated to system B. Since the pod namespace is private, there is no possible naming conflict. However, migrating even this simple program without pods could be problematic. Without pods, the PID namespace is global on a system. If system B already had a process running with PID 172, the child process in the above program would need to be assigned a new PID when it moves from system A. The parent process, however, expects the child process to have a PID of 172. When it calls `waitpid(iChildPID)`, it will be waiting for the wrong process.

Second, the private namespace masks out resources that are not contained within the pod, including processes outside of the pod. Pod namespace masking creates independence among elements that can migrate to avoid creating ties between components of the system that cannot be easily severed when an environment is migrated. A pod logically groups processes running on an operating system into three classes: processes inside the pod, special system processes outside the pod, and other processes outside the pod. Processes inside a pod appear to one another as normal processes that can communicate using traditional IPC mechanisms. Special system processes outside the pod that are consistently named across different machines, such as the init process in Linux, are viewable from within a pod but cannot interact with processes within the pod using IPC. In particular, the init process, or its equivalent system process that serves as the parent of orphaned children, is made viewable within a pod to serve as the parent of processes within a pod that have no parent process within the pod. Other processes outside a pod do not appear in the namespace and are therefore not able to interact with processes inside a pod using IPC mechanisms such as shared memory and signals. Instead, processes outside the pod can only interact with processes inside the pod using network communication and shared files that are normally used to support process communication across machines. Unlike IPC mechanisms, these communications mechanisms can avoid imposing host-specific dependencies that would limit the ability to migrate a pod.

4 Zap Architecture

Zap provides an architecture that supports the pod abstraction in the context of commodity operating systems. This is difficult to achieve because there are important mismatches and conflicts between the pod abstraction needed for process migration and current operating system design. For example, PIDs are determined by the host operating system which has no knowledge of pods, whereas pods require their own private PIDs for their respective processes. Processes running on the same operating system are free to communicate using IPC, whereas pods disallow such communication across pods to avoid creating host dependencies.

Zap provides pod functionality using commodity operating systems by inserting a thin virtualization layer between applications and the operating system. This virtualization layer is used to translate between pod namespaces and the underlying host operating system namespace, and protect processes within a pod from dependencies on processes outside the pod. We refer to pod resource names as *virtual names* and operating system resource names as *physical names*. Each pod virtual name corresponding to an operating system resource is mapped to an underlying operating system physical name. For example, a process is given a pod virtual PID that must be mapped to the physical PID used by the operating system. Zap virtualization is done by intercepting system calls and translating their arguments and return values as needed. When pod virtual names such as PIDs are passed to the operating system via system calls, Zap first translates those names to the corresponding operating system resource names then passes those physical resource names to the operating system. Similarly, when operating system resource names are returned via system calls to processes in a pod, Zap first translates those names to the corresponding pod resource names then passes those pod virtual names up to applications. Operating system resources without corresponding pod virtual names are masked out of the respective pod's namespace. Although Zap virtualization provides the flexibility of using virtual names that are distinct from the corresponding physical names, it does not preclude using virtual names that are the same as physical names. For instance, the virtual and physical PID for the init process are the same since the physical PID for init is always 1.

Zap virtualization is coupled with a checkpoint-restart mechanism to suspend, migrate, and resume pods and their associated processes. Zap uses a checkpoint-restart approach for the following four reasons. First, it is simpler to implement than other migration mechanisms such as demand paging. Second, it avoids leaving behind residual components after migration since all of the required state is in a checkpointed image that is simply moved for migration. Third, it enables flexible mechanisms for migrating the pod state, whether it be by sending it over a network or storing it on a disk that is moved from one place to another. Fourth, it enables pods to be checkpointed, suspended to secondary storage, and stored for future use.

To migrate a pod, Zap first suspends the pod by stopping all processes in the pod, saving the virtualization mappings, and saving all process state, including memory, CPU registers, open file handles, etc. The saved pod state can be digitally signed to avoid tampering and can then be moved to a new host in any number of ways. Zap defines an abstract I/O interface that is used for checkpointing and restarting a pod. The I/O interface has been defined specifically to allow only sequential writing (for checkpoint) and reading (for restart) of the image data. This ensures that checkpoint-restart can occur through the largest set of mediums, including ones which may not necessarily support two-way communication or which have extremely slow random-access speed. This allows data to be streamed to and from a tape or a network socket during migration, enabling pipelining of checkpoint-restart operations. On the new host, Zap resumes the pod by first restoring the pod virtualized environment, then restoring processes in a stopped state. Zap must then create necessary virtualization mappings pertaining to the stopped processes, such as remapping the virtual PID of a process to its new physical PID on the host. Finally, Zap enables the processes to continue executing in the restored pod environment.

The Zap architecture consistently applies the principles of pod namespaces, virtualization, and migration to a complete set of operating system resources. Table 1 provides an overview of how pod principles are applied to different resources. Section 4.1 discusses process identifiers and IPC resources. Section 4.2 considers memory resources. Section 4.3 focuses on file system resources. Section 4.4 introduces device resources. Section 4.5 overviews network resources. Section 4.6 describes pod administration and how pods are created and used.

Resource	Virtual Names	Migration State
Process State and IPC	Process and group IDs, IPC keys and IPC IDs	Process and group IDs, CPU registers, signal handlers, IPC keys and state, pipes and written, but not yet read by peer data in pipes
Memory	Memory addresses (already supported by OS)	Memory mappings and contents of data pages
File System	Directory structure, per-pod private directory, per-pod `/proc`	Opened files which have been unlinked, directories and files not mounted on a network file system
Devices	Per-pod `/dev`	Device names and state (device specific)
Network	Per-pod IP address, virtual network connections	Socket state, remote endpoint and virtual address pairs, queued but unsent data at the transport layer

Table 1: Pod principles applied to resources

4.1 Process IDs and IPC Virtualization and Migration

For each pod, Zap provides unique namespaces for process resources, including PIDs and keys for IPC mechanisms such as semaphores, shared memory, and message queues. Values from within these spaces are assigned the same way that they are by the operating system and are maintained consistently across migration.

Zap virtualization employs two types of hash tables that can be quickly indexed to translate these resource names between private pod namespaces and the operating system namespace. One is a system-wide hash table indexed by physical identifiers on the host operating system that returns the corresponding pod and virtual identifier. The other is a per-pod hash table indexed by virtual identifiers specific to a pod that returns the corresponding physical identifiers. When a system call is made that uses one of the identifiers in question, Zap replaces all the parameters that refer to virtual identifiers for the current pod with physical identifiers. Zap then invokes the system call, and upon return from the call replaces all physical identifiers returned by the system with virtual ones by looking them up in the system-wide physical-to-virtual hash table.

Consider PID virtualization as an example. Zap provides a host system-wide process identifier hash table indexed by the physical process PID and a process identifier hash table for each pod indexed by the virtual process PID provided by the respective pod. System calls that manipulate PIDs are intercepted, including `getpid`, `getgid`, `fork`, `kill`, etc. In system calls like `getpid`, the returned physical PID is translated into a virtual one and in system calls like `kill`, the virtual PID argument is translated into the physical one before passing it on to the kernel for processing.

Since Zap intercepts system calls that would manipulate process resource identifiers, such as the call to attach to a particular area of shared memory or to send a signal to a process, Zap can trivially limit the successful calls to those that use valid identifiers within their context. Within a pod, a valid identifier is a virtual identifier that was created within the pod. Outside of the pod environment, the system call will reject any requests to physical resources which belong to pods. This disallows a process on the host operating system from creating dependencies between pods and the host operating system.

To migrate a pod and its processes, Zap checkpoints data that pertains to process resources which is then used to restart the pod after migration. The data that is saved includes Zap virtualization state contained in the hash tables, process state for processes in the pod, and state associated with IPC mechanisms used by processes in the pod. Process state saved includes CPU registers, process credentials, process signal handlers and any pending signals, and process, group and session identifiers as well as the process's controlling terminal.

IPC mechanism state saved includes data associated with mechanisms used to access external entities, such as TCP/IP sockets, as well as IPC mechanisms used to communicate with other processes within the pod, such as Unix sockets, semaphores, shared memory areas, and unnamed pipes. Shared memory migration is only slightly different from typical memory migration discussed in Section 4.2. They are similar in that shared memory contains an address range as well as contents to fit within that range, however, shared memory requires a little more attention. First, a shared memory region, like many of the other IPC mechanisms is identified by a key that allows disparate processes to connect to the same resource. In addition, if multiple processes refer to the same shared memory region, they refer to the same contents as well. It is sufficient to save and restore the contents for the first process in the pod that uses the memory area, then for all other processes and mappings of that area, only a reference is needed.

When a pod is migrated and restarted on a target machine, Zap recreates the pod virtual environment and its processes. Zap creates new processes on the machine into which it restores the checkpointed process data. Zap keeps track of the new operating system identifiers corresponding to the new resources created and updates its system-wide and per-pod hash tables with this information to reconstruct the Zap virtualization state in the context of new operating system resources used by the pod on the target machine. Zap ensures that the pod environment is updated before enabling the processes in the pod to start executing again.

4.2 Memory Virtualization and Migration

Because memory state associated with processes is in general implicitly virtualized by the operating system, Zap only needs to provide virtualization support for mechanisms which allow a particular area of memory to be shared by multiple processes. Specifically, a memory location can be mapped to a file, shared as a result of process creation, and shared through IPC shared memory. File-mapped memory is handled implicitly by Zap's file namespaces, discussed in Section 4.3. Memory shared due to process creation is handled implicitly by automatically including child processes in the same pod as the parent so that they both share the same namespace. IPC shared memory is handled by pod virtualization as previously discussed in Section 4.1.

To migrate a pod, Zap checkpoints the memory areas allocated by processes within the pod. As discussed further in Section 4.3, Zap assures that the same view of the file system is available to a pod on whatever machine the pod is executed. As a result, Zap need not checkpoint the contents code segments that belong to an executable. This includes the text pages of the process as well as those of dynamically linked libraries in use by the process. Migration of text pages is accomplished through saving references to the executable files as well as the virtual memory addresses to which they are mapped.

When a pod is restarted, the checkpointed memory data needs to be restored. For efficiency reasons, whenever possible, Zap maps the checkpointed data directly to memory, allowing the pages to be read in as they are touched by the process. This can greatly improve the restart performance. If the checkpointed data is stored on a network file server, this can also reduce the network utilization of a process that will not necessarily touch all of its pages after restart.

4.3 File System Virtualization and Migration

To support migration of processes within pods, Zap must provide each pod with a consistent, location-independent view of the file system that is available on all hosts. One way this could be done would be to associate each pod with its own complete file system, which is migrated whenever the pod is migrated. Given that file systems can be many gigabytes, this would result in a substantial amount of state having to be migrated each time a pod moves from one machine to another making the approach impractical. Another way this could be done would be to have a global file system across all machines where a pod could be located. This removes the need to copy files from one machine to another since all files would be network accessible. However, ensuring a consistent global root file system across all machines is impractical as most machines are not configured this way in practice.

Zap takes a different approach to providing each pod with a consistent, location-independent view of the file system. Zap provides each pod with its own virtualized file system and corresponding private file system namespace. Zap then leverages widely used distributed file systems such as NFS [2] to store file data and mount such systems within the pod's virtual file system hierarchy. More specifically, when a pod is created or moved to a host, a private directory named according to a pod identifier is created on the host to serve as a staging area for the pod's virtual file system. Zap ensures that this directory is not accessible by processes on the host machine that are not in the given pod. Within this directory, the various network-accessible directories that the pod is configured to access will be mounted from a network file server. Minimally from a Unix-centric viewpoint, this set of directories would include `/etc`, `/lib`, `/bin`, and `/usr`. Each pod must also be configured with a `/tmp` that is private to the pod and is discussed further below. Zap then uses the `chroot` call to set the staging area as the root directory for the pod, thereby achieving file system virtualization with modest performance overhead. Zap virtualizes `chroot` to prevent processes within a pod from breaking out of their virtual file system environment. This approach takes advantage of distributed file systems to reduce file state that would need to be moved during migration without requiring a global file system across all host machines.

Zap can use network file servers to support many pods running on many machines at the same time. This is simple to do for files such as common executables which are used in the same way by processes in all pods. In some cases though, it is desirable and necessary to have files and directories that are specific to a given pod. For this purpose, Zap introduces private pod directories. A private pod directory is created when a pod is initially created and destroyed once the pod is finally destroyed. The private directory is only used by the given pod. The directory can be created on the network file server then mounted within the pod virtual file system as `/privatepod`. When a private pod directory is created, it may optionally be pre-populated with data from a template directory.

Private pod directories can be useful for supporting directories such as `/tmp` or `/local` that are typically local to a machine. It is important that such directories are private to avoid naming conflicts that would otherwise arise in the file system due to the way some legacy applications name files. For example, some server applications store their PIDs in the `/tmp` directory in a file with the server name as the filename and `.pid` as the file extension. If `/tmp` is shared by two pods that happen to both have instances of such a server running, a filename conflict will result. These conflicts arise due to processes in separate pod namespaces sharing the same file namespace and occur with directories such as `/tmp` that are typically local to a machine. Zap avoids this problem by storing `/tmp` and `/local` in the private pod directory when a pod and its virtual file system are created. Whereas `/tmp` is initially empty, `/local` can be populated by transferring over files from a template directory on the file server. Note that Zap does not have to use a network file server for such these private directories but could instead store them locally on the host machine as a subdirectory of the pod's virtual file system. However, this would require that the files be migrated as well when the pod is migrated.

Private pod directories can also be useful for allowing per-pod application configurations without having to duplicate the application file hierarchy. When some files or subdirectories used by a common application need to be specific to a given pod, these files can be easily configured as symbolic links to files in the respective private pod directories. For example, to install a web server that is available to all pods, an administrator could install the web server in a global `/usr/local/apache` directory, and make the conf directory within it a symbolic link to `/privatepod/apache/conf`. This will allow multiple pods to share one copy of the web server, which can be centrally managed and upgraded periodically to fix bugs and close up security holes, while each pod maintains its own configuration, allowing pods to point to log files and web pages anywhere on their file system.

In addition to network accessible files and private pod files, Zap must also consider special file systems such as the proc file system [27] in `/proc` and devices in `/dev`.

We briefly discuss /proc here and defer devices to Section 4.4. Each pod is given its own /proc by creating a special per-pod directory, specifically /proc/zap/pods/<pod_id>/proc, under the proc file system of the host machine and loopback mounting that area as /proc in the pod's virtual file system. The per-pod directory registers its own set of file operations with the proc file system for accessing files and directories instead of using the generic operations used when accessing files in /proc directly from the host. These pod /proc file functions are similar to the generic /proc functions except that they translate as needed to return system information in the context of the pod namespace. For example, the process directories in the pod /proc are listed by virtual pod PIDs instead of physical PIDs. All of the information in the Zap-specific area of /proc is created dynamically by combining pod virtualization information with system information from the operating system.

When a pod migrates, Zap flushes all cached data to disk and saves and restores the information it needs to reconstruct the pod virtual file system, including a list of all files opened by the processes within a pod and the access rights with which the files were opened. Zap in general does not need to save file contents because it leverages the use of a distributed file system to make the files available at the machine where the pod is resumed. Dynamically generated files in /proc also do not need to be saved since they can be recreated at the machine where the pod is resumed. In environments where a pod cannot access distributed file systems from all locations, Zap could easily be extended to package up the contents of the file system along with the checkpointed image of the pod.

Zap does checkpoint the contents of files that have been opened by a pod process and have been subsequently unlinked. This is because as soon as the pod processes are checkpointed, opened files are closed and the file system will free up the inodes associated with the files, losing the data that they contained. Unlinked files will no longer exist and will not be available when a pod is restarted. To address this problem, Zap checks the reference counters of inodes belonging to files opened by processes in a pod. If a pod has an unlinked file open, Zap saves the contents of the file and recreates the file when the pod is restarted, once again unlinking it after it has been opened.

4.4 Device Virtualization and Migration

Zap provides each pod with its own virtual /dev directory to provide a framework for supporting device virtualization and migration. Creating a unique /dev directory for a pod helps achieve two goals: first, it ensures that the pod cannot accidentally use any host-specific devices which may be difficult to migrate because they may be in an awkward interim state (such as the CD recording device while recording), or impossible to migrate because they are unavailable elsewhere (for example, just because one host is attached to an electron microscope, one cannot assume that all hosts will be attached to one). The second goal it achieves is naming resolution for certain files, for example, virtual tty names. When the system assigns a new tty to a process, it merely selects the next available tty number (for example, /dev/pts/2 or /dev/tty02). Without virtualizing the /dev namespaces, there would be no guarantee that the particular tty will be available for a given pod at its new location.

Each pod is given its own /dev by creating a special per-pod directory on the host machine and loopback mounting that area as /dev in the pod's virtual file system. Zap employs a device-specific plugin for each device, which registers a particular device within the virtual /dev directory and provides appropriate support for the given device. Device support needs to be addressed on a per-device basis; full migration support of devices is a difficult problem that is beyond the scope of this paper. Devices that are not explicitly supported by Zap are not included in a pod /dev directory, preventing processes within a pod from accessing them.

Zap defines three types of device support that could be provided as emulation, virtualization, and non-migratable. Emulation could be used to emulate a device in software. For example, a virtual console could be created and registered as /dev/console. When data is written to /dev/console, it could be redirected to a pre-defined text file accessible within the pod. Virtualization could be used to utilize an equivalent device on the host system from within the pod. For example, a virtual /dev/audio could be created. As the virtual /dev/audio is accessed, the virtual device driver would make note of any configuration changes requested of the audio device and pass them on to the host's audio device. When the pod is migrated to a new host, the audio device on the original host is closed and reset and the audio device on the new host is opened and configured with whatever state changes were invoked previously. Finally, non-migratable device driver could be created that passes all requests to the device on the host machine, but disallows migration so long as the device is in use. For example, when using a CD recording device, migrating in the middle of recording a CD would result in a CD with half its contents on being recorded on the original host and the other half on the new host. A non-migratable device driver could simply cause the pod migration to fail until the CD recorder device has been closed.

Device virtualization and migration greatly depend on the device and its capabilities. Currently, Zap explicitly supports pseudo-terminal devices, which allow one to log in remotely as well as have multiple terminals open through such programs as `xterm`. Zap provides virtual ttys which require that, when a tty is opened, rather than the next available virtual tty for the host being returned, the next available virtual tty for the pod is returned. Upon migration, the migration process specifies the desired virtual tty number and the virtualization system will automatically map between the host's tty number and the pod's tty number.

4.5 Network Virtualization and Migration

Zap is designed to support migration of unmodified network applications running in pods using the existing network infrastructure without any modifications. Toward this end, Zap provides mechanisms to address three key issues: (1) enabling remote systems to locate and communicate with processes in the pod, (2) exposing application layer network interfaces to support persistent communication in the presence of migration, and (3) preserving consistent state at the transport layer to maintain an application's open connections persistently even after the application migrates.

To enable remote systems to locate and communicate with processes in the pod, Zap allows each pod to be assigned an external IP address that can be known to entities outside of a pod. The address is distinct from the IP address of the host machine where the pod is currently located. This external address changes as the given pod moves from host to host in the same way that a host machine's IP address changes when it changes network locations. The external IP address can be a routable one to enable a pod to provide network services such as a web server that can be accessed from an outside-initiated connection.

To allow a pod's network services to be accessed even if its external address changes due to migration, Zap leverages dynamic DNS [40] to maintain a name-to-IP relationship so that a pod's network services can be accessed by the same name even after migration. To avoid downtime due to migration, Zap can install a temporary proxy process just before a pod migrates, stall current and pending connections for a few seconds without negative effect, and then restart them once the pod is resumed on the new host machine. The number of incoming connections to the proxy will drop off as the TTL expires and when it gets below a defined threshold, the proxy terminates. The proxy only needs to run for a short time with a TTL of a few minutes, which does not adversely affect DNS caching performance [20].

To provide network interfaces for applications in a pod to support persistent communication in the presence of migration, Zap distinguishes between the external IP address and internal IP address perceived by a process running within the pod. Pods only allow applications within a pod to perceive the internal IP address directly. Pods can provide two types of network interfaces for determining the internal address seen by processes running within the pod: transient and persistent. Transient network interfaces assign the internal address equal to the current external IP address, which exposes the movement of a pod to applications within the pod. This is the default behavior, which works fine for most network applications because they do not require a fixed IP address for their connections to persist and function correctly even after a pod migrates. This default behavior also supports context-aware applications such as network discovery tools that need to know about IP addresses changes to operate correctly.

Persistent network interfaces assign the internal address equal to a static value that does not change due to pod migration. This is useful for applications that require the IP address that they see to remain unchanged in order for their open network connections to continue to function correctly. Although such applications are in the minority, one such application that is widely-used is FTP, which explicitly checks the source and destination IP addresses used for consistency for its protocol interaction. The static value that is used can be assigned in almost any manner. By default, the internal address seen by a connection is equal to the external IP address in use when the connection was first opened and persists for the lifetime of the given connection. Alternatively, the internal address can be statically assigned to a predefined value that is constant across all connections. This would be useful for a multi-homed web server, which is configured with multiple IP addresses and performs a different action depending on the IP address from which a request is made. By having predefined internal addresses, the web server can be started in a pod with the same configuration regardless of the host machine on which the pod is located. This can simplify web server administration for web hosting providers running web servers on a cluster of homogeneous machines.

To maintain an application's open connections persistently even after the application migrates, Zap needs to ensure that the network state perceived by the transport protocol for a given connection remains the same before and after migration, even though the pod's external IP address needs to change when its location changes. To address this problem, Zap incorporates a novel virtual networking mechanism that transparently supports persistent open end-to-end connections among migrating pods. The mechanism is based on the Virtual Network Address Translation (VNAT) architecture presented in [39], tailored specifically for Zap.

The idea behind VNAT is surprisingly simple. A virtual address, rather than a physical address, is introduced to identify a pod for its end-to-end connections. We use the term "address" loosely to refer to both an IP address and port number, both of which are virtualized by VNAT. To send data over the connection, the virtual address is then translated into an appropriate physical address after packets leave the transport protocol and before they are injected into the network. Conversely, to receive data over the connection, a physical address is translated back into the corresponding virtual address before packets are returned to the transport protocol layer. Using these connection virtualization and translation mechanisms, a pod can migrate from place to place without changing the network connection state visible to the transport protocol. Note that Zap network virtualization differs somewhat from other resource virtualization as Zap must virtualize resources not just below the application layer, but below the transport layer as well.

Connection virtualization works by intercepting system calls for connection setup requests from the application to the transport protocol and replacing relevant physical addresses with virtual addresses. The result is that the transport protocol stack on both the client and the server will perceive a virtual connection with virtual addresses rather than a physical connection with the actual physical addresses of the machines. This virtual connection identification will stay unchanged for the life of the connection no matter where the pod containing the client or the server moves.

Connection translation works by intercepting packets below the transport protocol layer and translating the virtual addresses in the packet headers to physical addresses. A packet with a virtual address header is not routable on the physical network. Using connection translation, Zap translates a packet with a virtual address header sent by the client transport protocol into a packet with a physical address header so it can reach the intended server. Although this is similar to Network Address Translation (NAT), a key difference is that connection translation works entirely within the endpoint without introducing any connection state inside the network. Therefore connection translation does not suffer from many pitfalls experienced by traditional NAT [15, 16, 37].

To migrate a pod with open network connections, Zap checkpoints network state pertinent to its open network connections including standard operating system states such as socket structures and transport protocol states such as TCP PCB, as well as connection virtualization and translation states created by Zap. During the restart, the standard operating system states and transport protocol states are simply restored to their original values and the restarted pod can trivially locate the server location using the existing connection state. However, connection virtualization and translation states need to be updated on both endpoints to reflect the mapping between the (constant) internal address and the new external address of the migrated pod. When a connection endpoint resumes after migration, Zap notifies the endpoint on the other side of the connection that the migrated endpoint is at a new physical address. Both endpoints of the connection then update their virtualization state so that their virtual address pairs map to the new physical address pair. The protocol used to update the endpoints is detailed in [39]. Note that the virtual connection perceived by the transport protocol stays intact across the migration and the transport layer is completely unaware of the change of the underlying physical address of the client. So with the addition cost of translating a virtual connection to and from a physical connection, Zap will seamlessly migrate a transport end-to-end connection regardless of where the client moves.

Zap selects the virtual address for a connection to be the same as the current physical address for the connection, which corresponds to the current external IP address of the pod. This choice of virtual address provides two key advantages. First, it eliminates the need for the client and server to exchange their virtual addresses at connection setup time. Second, it eliminates connection translation overhead for connections that are not migrated. As a result, no translation overhead will ever be imposed on a connection so long as the pod does not move. After a pod migrates, only existing connections that have migrated along with the pod will incur connection translation overhead. New connections from the migrated pod will not incur such overhead since the virtual address used for the new connections will always be based on the current external address of the pod.

Zap distinguishes between the virtual address seen by the transport layer and the internal address seen by applications running within a pod. Since Zap intercepts system calls that return network addresses to applications, Zap can return any network address that it chooses. This makes it simple to support both transient and persistent network interfaces as discussed previously. For transient network interfaces, Zap returns the underlying external IP address to applications. For default persistent network interfaces, Zap propagates the virtualization up to applications and returns the virtual address seen by the transport layer when a connection was first established; for persistent network interfaces with statically assigned internal addresses, Zap simply remembers the static assignment and returns that value as the internal address.

In addition to providing connection persistency between applications running in pods, Zap can also provide support for preserving connections between pods and traditional processes running outside of pods assuming that such connections are made through a proxy. The network virtualization and migration mechanism of Zap can be installed separately from pods on a proxy. Zap network virtualization preserves a connection between a pod and such a proxy the same way it preserves a connection between two pods. When the pod migrates, the connections between the pod and the proxy continue to be "spliced" with the connections between the proxy and the legacy applications behind the proxy. Since the proxy doesn't detect the movement of the pod due to Zap network virtualization, there is no need to "switch" the connections between the pod and the proxy. As a result, the complete proxied connections between the pod and traditional processes without pods are migrated without any modifications to applications or system environments without pods.

4.6 Pod Administration and Usage

Zap enables any user to create a pod, either explicitly or by incorporating pod creation into system utilities such as `login` to encapsulate a user's computing session within a pod. Once a pod is created, access to the pod is controlled by an access control list (ACL). Only the host system administrator on the system where the pod is currently executing or a user on the ACL are allowed to manipulate the pod, including suspending and resuming a pod for migration purposes. The five primary commands provided by Zap for users and system administrators to create and manipulate pods are:

create_pod enables a user to create a new pod with various options that can be specified in a pod configuration file. These options include the network configuration and file system configuration for the pod, what applications if any should be launched once the pod is created, and access control permissions for the pod. `create_pod` assigns a numerical identifier to the pod and creates a corresponding entry for the pod in a list maintained by Zap that contains a list of pods currently running on the host machine and associated information about each pod. The pod identifier may change when a pod migrates to another machine. Pod creation does not nest, so that creating a pod from within an existing pod will create a new pod on the host machine. Some examples of how `create_pod` can be used include: with `login` to create a user session in a pod, with `/etc/init.d` for automatically starting up services like a web server in a pod, or with `inetd` to spawn incoming connection handlers into a new pod, such as creating a separate pod for each `telnet` session.

kill_pod takes a pod identifier and terminates the respective pod, killing all processes in the pod, freeing all associated resources, and removing the pod itself from the system.

addproc_pod takes an executable and a pod identifier and creates a new process running the given executable in context of the respective pod. Note that `addproc_pod` only creates a new process in the pod; it will not move an already existing process into the pod to avoid creating naming conflicts. To add a process to a pod, Zap must first create a new process. In Unix systems, process creation is done using `fork`, which creates a child process that is a copy of the parent by allocating a kernel process structure and populating it with information from the parent. Zap creates a process in a similar fashion but does not use the same kind of information from the parent in the new process to ensure that there are no dependencies on the parent process or the host system or pod in which the parent process resides. Instead, Zap takes several steps to avoid creating any such dependencies in the new process, including setting its parent process to `init`, making it the process group leader, and relinquishing the control terminal.

Because adding a process to a pod is done in a system call, all of the necessary steps can be done in the kernel before the process is made runnable. In particular, the executable that `addproc_pod` specifies to run is overlaid on the new process before returning from the system call, not as a separate `exec` system call. Furthermore, the executable that `addproc_pod` executes is specified in the context of the virtual file system of the pod into which the process is being added, not the file system of the environment from which the `addproc_pod` command was issued. Finally, `addproc_pod` also creates an entry for the new process in the pod's process list. Although special care must be taken in adding a process to a pod, process creation inside of a pod is simply done in the normal manner with a created child process inheriting the attributes of its parent. Once a process has been added to a pod, all its operations occur inside of the pod, and all of its children will also be created inside of the pod.

suspend_pod takes a pod identifier and filename, stops the pod and all of its processes, checkpoints the state of the pod to the respective file.

resume_pod takes a filename for a file that contains a checkpointed pod and restarts the pod and its processes starting at the point at which the pod was checkpointed. `resume_pod` assigns a new numerical identifier to the pod.

To simplify administration, Zap provides a view of the host machine outside of the context of pods that shows everything running on the given machine in the same manner used in existing operating system environments. An administrator can access the host machine directly to obtain this administrative view. Processes within pods are viewable from this administrative view, but they cannot be accessed from this view using traditional IPC mechanisms to avoid creating host dependencies. We note that pods introduce interesting security considerations, but due to space constraints, these security issues are not discussed here.

5 Implementation

The Zap architecture was designed to provide transparent process migration while minimizing changes to the operating system by leveraging the loadable kernel module interface available in many commodity operating systems. We have implemented a Zap prototype as a Linux kernel module. Our implementation builds on previous work of one of the authors on a Linux kernel module called CRAK [42] that provided a restricted process migration mechanism but did not support the general pod abstraction. Our Zap implementation can be dynamically loaded on a running Linux system to provide Zap functionality without modifying, recompiling, or reinstalling the Linux kernel. We highlight some key mechanisms that were used for implementing Zap virtualization and migration in a kernel module.

Zap virtualization was largely implemented by providing a mechanism for intercepting system calls to translate between pod and operating system namespaces. Intercepting a system call within a Linux module is fairly simple. The module need only replace the appropriate system call handler pointer in the system call table by a pointer to the new system call handler. In order to invoke the previous system call handler, the new handler need only call the old function pointer. This results in a small amount of additional overhead due to the extra procedure call. For simple system calls like `getpid`, the only extra cost beyond the procedure call is a hash table lookup for translation. For more complex system calls like `fork`, Zap also needs to allocate pod-specific structures for keeping track of necessary process state for processes running in a pod. Note that when pods are running, system calls are also intercepted for processes running outside of any pod to check to make sure that those processes are not attempting to communicate directly with processes in a pod using local mechanisms.

Our Zap virtualization implementation also uses NFS for file system virtualization and Linux *netfilter* for network virtualization. Zap creates the virtual file system of a pod by mounting various NFS mount points from a file server for pods to a staging directory on the local machine. Zap uses the netfilter system in the Linux 2.4 series kernel, which is a packet filtering and mangling system [34]. Netfilter instruments the IP protocol stack at well-defined points during the traversal of the stack by a packet. It provide hooks that invoke user-registered functions to process the packet at these well-define points. Zap uses these hooks for source and destination address translation on both incoming and outgoing network traffic.

Zap migration was implemented by providing mechanisms to read and write kernel state to checkpoint and restart pods. Since kernel modules run in kernel mode, a Zap kernel module will have the necessary privileges to read and write kernel state that must be saved and restored for checkpointing and restarting pods. However, locating the process-related data structures that needed to be saved and restored was more difficult because Linux does not export all of the kernel structures to kernel modules. As it turns out, when the Linux kernel is built and distributed, there is a file called `System.map` which contains a list of all symbols, both exported and not, and their locations in the kernel memory space. Therefore, Zap simply queries this file to identify the addresses of structures which contained data to be checkpointed for the processes, as well as functions which were integral to the restoration of the processes.

One interesting decision we encountered while developing Zap's migration mechanism was whether the kernel structures should be saved in their native format or the individual elements from the structures saved. We decided to save the individual elements to enable Zap to migrate processes across minor version changes of kernels, even if the layout of their structures change slightly. In order to achieve cross-version migration, Zap must be able to translate the process and kernel state from the format provided by the source host to the format required by the target host. Not only must Zap be able to ensure compatibility between the versions, but it must also be capable of populating the target host with whatever values that were not provided by the source host. Although we have successfully used Zap to migrate pods across Linux kernels with minor version differences, a more complete examination of this issue is the subject of future work.

6 Experimental Results

We present some experimental results using our Linux Zap prototype on various applications. We conducted our experiments on a trio of IBM Netfinity 4500R machines, each with a 933 MHz Intel Pentium-III CPU, 512 MB RAM, 9.1 GB SCSI HD, and 100 Mbps Ethernet connected to a 3Com Superstack II 3900 switch. One of the machines was used as an NFS server from which directories were mounted to construct the virtual file system for a pod. All of the machines were installed with the RedHat Linux 7.1 distribution and the Linux 2.4.10 kernel.

Since VMMs have been proposed for migration, we also performed our experiments with VMware Workstation 3.2 for Linux with the same RedHat distribution and Linux kernel running in a VM. This provides a conservative comparison of our unoptimized prototype against a tuned commercial product. Unless otherwise indicated, the VM was configured with raw disk mode, bridged networking, and the recommended 384 MB of memory. While VMware did not allow us to configure the VM with the same memory size as the host RAM, we ensured that memory size was not a limitation for our experiments.

Section 6.1 describes some simple experiments to measure the cost of Zap virtualization compared with a vanilla Linux system and VMware. Section 6.2 describes examples of how pods can be used to provide mobile thin-client computing sessions and web servers, and measures the cost of migrating these sessions using Zap versus VMware.

6.1 Zap Virtualization

To measure the cost of Zap virtualization, we used a range of micro benchmarks and application benchmarks to measure both individual system call performance as well as real application performance. The nine benchmarks we used are described in Table 2. All of the benchmarks measure the time it takes to run the respective benchmark. volano is VolanoMark 2.1.2, an industry standard Java chat server benchmark configured in accordance with the rules of the Volano Report [4], but reports the average time per message transferred rather than the message transfer rate.

Name	Description	Linux
getpid	`getpid` run in a loop 10000 times, measure average iteration time	352 ns
shmget +shmctl	IPC shared memory segment holding an integer is created and removed	42 µs
semget +semctl	IPC semaphore variable is created and removed	19 µs
fork+ exit	process forks and waits for child which calls exit immediately	111 µs
fork+ execve	process forks and waits for child to run C program that prints "hello world" then exits	1811 µs
fork+ /bin/sh	process forks and waits for child to run `/bin/sh` to run C program that prints "hello world" then exits	7963 µs
volano	VolanoMark 2.1.2 using Java 2 Runtime Environment SE 1.4.1	219 µs/ mesg
make	Linux kernel compile with up to ten processes active at one time	440 s
apache	Netscape browser downloads Java-script-controlled sequence of 54 web pages from Apache 2.0.35 web server	16 s

Table 2: Application benchmarks

To obtain accurate measurements, we rebooted the systems between measurements and directly used the TSC register [3] available on Pentium CPUs to record timestamps at the significant measurement events. The average cost of each timestamp was 32 ns. The files for these benchmarks were stored on the NFS server for all of our experiments to provide a consistent comparison. We measured the performance of these benchmarks on four different Linux 2.4.10 system configurations:

- Linux - benchmarks are run on a vanilla Linux system to measure baseline system performance.
- VMware - benchmarks are run on vanilla Linux (guest OS) inside a VM on a vanilla Linux system to measure performance using a VM.
- With Pod - benchmarks are run on a Linux system with Zap installed and a pod created to measure performance on the host outside of a pod.
- Inside Pod - benchmarks are run in a pod on a Linux system with Zap installed to measure performance inside of a pod.

Table 2 shows the results of running the nine benchmarks on the vanilla Linux system. Figure 1 shows the results of running the benchmarks on the other three system configurations, the results normalized to the vanilla Linux system with the value one representing the normalized vanilla Linux results. Since all benchmarks measure the time to run the benchmark, a small number is better for all benchmark results in Figure 1.

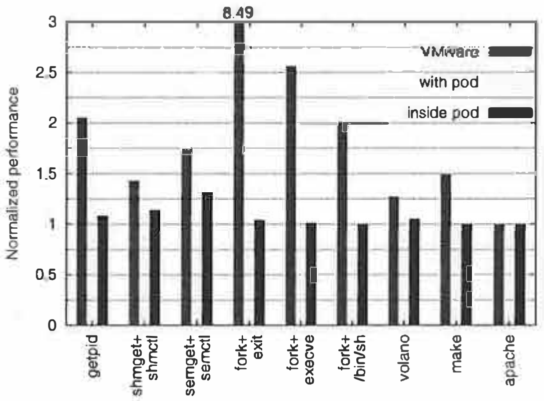


Figure 1: Virtualization cost

The results in Figure 1 show that VMware performs the worst on all of the benchmarks. The simple getpid benchmark takes more than twice as long using VMware compared to vanilla Linux. fork+exit gives the worst performance on VMware, running more than eight times slower than vanilla Linux. VMware does better on the benchmarks that are not dominated by system calls, but still runs 100% slower on fork+/bin/sh, 20% slower on volano, 50% slower on make compared to vanilla Linux. The only benchmark on which VMware does not perform worse is apache, where a web browser on another machine downloads and displays a sequence of 54 web pages from an Apache server running inside VMware. For apache, all of the system configurations delivered essentially the same performance.

Figure 1 shows that Zap virtualization overhead is quite small, especially compared with using a virtual machine monitor like VMware. When running inside a pod, Zap overhead for the simple system call getpid benchmark was only 8% compared to vanilla Linux, reflecting the fact that Zap virtualization for these kinds of system calls only requires an extra procedure call and a hash table lookup. The most expensive benchmark for Zap was semget+semctl, which took 32% longer than vanilla Linux. The cost reflects the fact that our untuned Zap prototype needs to allocate memory and do a number of namespace translations. semget+semctl and shmget+shmctl both take 6 μs longer with Zap than vanilla Linux, but this accounts for a higher percentage of time for semget+semctl because it takes roughly half as much time overall. volano overhead with Zap is 5% more than vanilla Linux due to the overhead of using `clone` to generate high thread counts. There was no additional overhead for running make inside a pod compared to running on vanilla Linux.

Figure 1 also shows that the cost of running pods is also quite small for processes that are running on a host system outside of pods. With pods present, Zap must intercept system calls made by processes outside pods to ensure that they do not attempt to manipulate processes inside pods. The overhead of this pod protection was less than 10% compared to vanilla Linux for the benchmarks dominated by system call cost, namely getpid, shmget+shmctl, semget+semctl, and fork+exit. More importantly, there was no difference in overall performance between running with pods and vanilla Linux for any of the other larger application benchmarks that we tested.

6.2 Zap Migration

To illustrate how Zap can be used for different applications and measure the cost of migration using Zap, we used two different applications listed in Table 3, a VNC thin-client computing user session and a web server. The VNC [32] thin-client computing user session provides an example of how Zap can be used to provide mobility of a user's computing session. Table 3 shows that eleven legacy and network X applications were run as part of the VNC session for our experiments. The apache session provides an example of how Zap can be used to provide web server mobility. We encapsulated each of the two sessions in a pod and measured the cost of suspending each pod and resuming it on another host machine, both in terms of the time to checkpoint and restart the pods and in terms of the amount of state that needs to be saved for migration. For comparison, we also measured the cost of suspending and resuming these two application sessions on the same machine using VMware. We did not actually migrate the sessions using VMware because it does not support migration of networked applications.

The results of migrating the application sessions using Zap and suspending and resuming them using VMware are shown in Figure 2. For these experiments, the ses-

Name	Applications
VNC	Xvnc 3.3.3 - VNC virtual X server twm - window manager Netscape communicator 4.76 - web browser telnet - telnet client inside `xterm` window, connected to another machine on the same LAN xterm+bash - shell running in `xterm` window xview - image viewer w/ 13 KB GIF loaded xcalc - X-based calculator xclock - analog clock xman - X-based man page browser xelvis - `vi` text editor w/ 870 byte text file loaded xpdf - PDF viewer w/ 293 KB 14-page file PDF loaded
apache	Apache 2.0.35 - web server, default number of worker processes used

Table 3: Application sessions

sions were checkpointed to and restarted from an image on the local disk. For VMware, we ran experiments with two VM configurations, one with 128 MB of memory and the other with 384 MB of memory. In all cases, the resulting checkpoint image was always a little more than the memory size given to the VM, regardless of what applications were running. The time to suspend the sessions grew disproportionately with the memory size given to the VM, taking about 2 seconds per session for a 128 MB VM, but more than 25 seconds per session for a 384 MB VM.

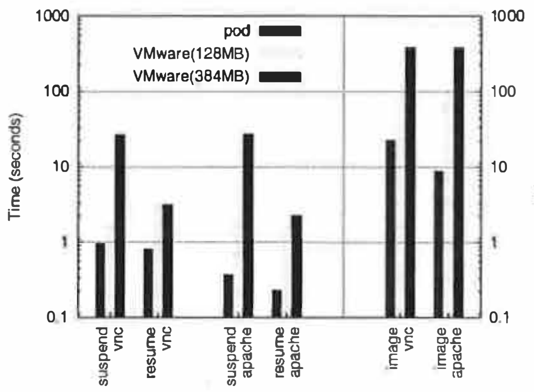


Figure 2: Migration cost

Our results show that Zap saves much less state and is much faster than VMware in suspending and resuming a running application session. Figure 2 shows that checkpoint and restart times for each pod were less than a second. Checkpointing the VNC pod took 963 ms and resulted in 23 MB of image data, whereas checkpointing the apache pod took 373 ms and resulted in 9 MB of image data. Memory contents accounted for over 99% of the checkpoint image file sizes, but only grow with the applications actually used, as opposed to VMware in which the sizes grow with the RAM allocated to the VM. If the medium over which the checkpoint images are transferred for migration were slow or somehow limited in capacity, the checkpointed pod images could be compressed further using gzip, resulting in a 4.6 MB VNC pod image and a 0.9 MB apache pod image. Restarting the pods was slightly faster. Restarting the VNC pod took 811 ms and restarting the apache pod took 231 ms. The restart times are faster than checkpoint times in part because parts of the image files are mapped directly to memory during restart and are loaded by the operating system as they fault, whereas the checkpoint process needs to save all state.

We further analyzed the amount of time that Zap spent checkpointing and restarting each application within the VNC pod. The time to checkpoint an application in a pod was generally bound by the resulting image size. The application with the largest checkpoint time was Netscape, which had 10 MB of memory contents which needed to be saved. Most other application checkpoint times varied with the amount of memory pages which needed to be saved, suggesting that the checkpoint times are primarily I/O bound. However, the telnet application restart time accounted for a disproportionate amount of the time to restart the VNC pod, especially given its modest contribution of 363 KB to the 23 MB pod image size. The reason for this is because restarting this network application required a round-trip message to the remote end of the connection to inform it of the new location and set up translation rules. Nevertheless, the overall cost of restarting telnet in its `xterm` window was still modest at only about 140 ms. This time is not accounted for in the VMware measurements since the VMware could not migrate the VNC session and maintain the telnet connection.

Our results for migrating pods running realistic applications show that pod migration costs were modest overall, with subsecond checkpoint and restart times for the VNC and apache pods. More importantly, our results demonstrate the ability of Zap to migrate legacy applications without modification, including graphical X applications and networked applications such as telnet.

7 Conclusions and Future Work

Zap is the first system that we are aware of that provides transparent migration of legacy and networked applications across machines running independent operating systems without requiring any changes to the operating systems. Zap achieves this behavior by leveraging loadable kernel module technology and introducing a thin virtualization layer that decouples applications from host dependencies in the operating system. Zap introduces and supports a pod abstraction, which encapsulates groups of processes in a virtualized environment that can be migrated as a unit. We have implemented Zap as a Linux kernel module to demonstrate the viability of our approach. Our experimental results on real applications using our Linux Zap prototype show that Zap can provide general-purpose process migration functionality with low overhead. We hope that Zap will provide a useful tool and building block for exploring the benefits and applications of migratable computing environments.

Zap raises a number of interesting follow-up research areas. First, Zap raises many interesting questions, both in terms of security mechanisms that should be implemented within Zap itself as well as security mechanisms and policies that should be considered for systems hosting pods. Current security schemes do not generally take into account process migration; as such, more work should be done to properly understand the issues and how they may be addressed. In addition, process migration is most beneficial when used under the appropriate circumstances. This raises the question of when to migrate a pod, and which pod to migrate. Finally, support for additional devices remains an open question because each device has its own requirements. As such, additional study into how common devices should be handled is warranted.

8 Acknowledgments

Eric Brewer, our paper shepherd, Brian Schmidt, and Erez Zadok provided many helpful comments on earlier drafts of this paper. Shaya Potter contributed to Zap design discussions and parts of the Linux Zap implementation. This work was supported in part by NSF grants EIA-0071954 and ANI-0117738, an NSF CAREER Award, and an IBM SUR Award.

9 References

[1] *http://www.vmware.com*, VMware, Inc.

[2] *NFS: Network File System Protocol Specification*, RFC1094, Sun Microsystems, Inc., March 1989.

[3] *Using the RDTSC Instruction for Performance Monitoring*, Pentium II Processor Application Notes, Intel Corporation, 1997.

[4] *The Volano Report*, Volano LLC, December 2001. http://www.volano.com/report

[5] K. Amiri, D. Petrou, G. Ganger, and G. Gibson, *Dynamic Function Placement in Active Storage Clusters*, Technical Report CMU-CS-99-140, School of Computer Science, Carnegie Mellon University, June 1999.

[6] Y. Artsy, Y. Chang, and R. Finkel, *Interprocess Communication in Charlotte*, IEEE Software:22-28, January 1987.

[7] A. Barak and R. Wheeler, *MOSIX: An Integrated Multiprocessor UNIX*, Proceedings of the USENIX Winter 1989 Technical Conference, pp. 101-112, San Diego, CA, February 1989.

[8] P. Bhagwat, C. Perkins, and S. K. Tripathi, *Network Layer Mobility: an Architecture and Survey*, IEEE Personal Communication, 3(3):54-64, June 1996.

[9] T. Boyd and P. Dasgupta, *Process Migration: A Generalized Approach Using a Virtualized Operating System*, Pro-

ceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002), Vienna, Austria, July 2002.

[10] J. Casas, D. L. Clark, R. Conuru, S. W. Otto, R. M. Prouty, and J. Walpole, *MPVM: A Migration Transparent Version of PVM*, Computing Systems, **8**(2):171-216, 1995.

[11] D. Cheriton, *The V Distributed System*, Communications of the ACM, **31**(3):314-333, March 1988.

[12] F. Douglis and J. Ousterhout, *Transparent Process Migration: Design Alternatives and the Sprite Implementation*, Software - Practice and Experience, **21**(8):757-785, August 1991.

[13] I. Foster and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit*, Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, Lyon, France, August 1996.

[14] A. Grimshaw and W. Wulf, *The Legion Vision of a Worldwide Virtual Computer*, Communications of the ACM, **40**(1):39-45, January 1997.

[15] T. Hain, *Architectural Implications of NAT*, RFC2993, IETF, November 2000.

[16] M. Holdrege and P. Srisuresh, *Protocol Complications with the IP Network Address Translator*, RFC3027, IETF, January 2001.

[17] D. B. Johnson and C. Perkins, *Mobility Support in IPv6*, draft-ietf-mobileip-ipv6-16.txt, IETF, March 2002.

[18] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek, *Mobile Computing with the Rover Toolkit*, IEEE Transactions on Computers, **46**(3):337-352, March 1997.

[19] E. Jul, *Migration of Light-weight Processes in Emerald*, IEEE Technical Committee on Operating Systems Newsletter, **3**(1):20-23, 1989.

[20] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, *DNS Performance and the Effectiveness of Caching*, Proceedings of ACM SIGCOMM Internet Measurement Workshop, pp. 153-167, San Francisco, CA, November 2001.

[21] M. Kozuch and M. Satyanarayanan, *Internet Suspend/Resume*, Fourth IEEE Workshop on Mobile Computing Systems and Applications, Calicoon, NY, June 2002.

[22] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, Technical Report #1346, University of Wisconsin Madison Computer Sciences, April 1997.

[23] D. A. Maltz and P. Bhagwat, *MSOCKS: An Architecture for Transport Layer Mobility*, Proceedings of the IEEE INFOCOM'98, pp. 1037-1045, San Francisco, CA, 1998.

[24] D. Milojicic, F. Douglis, and R. Wheeler, *Mobility: Processes, Computers, and Agents*, Addison Wesley Longman, February 1999.

[25] S. J. Mullender, G. v. Rossum, A. S. Tanenbaum, R. v. Renesse, and H. v. Staveren, *Amoeba – A Distributed Operating System for the 1990s*, IEEE Computer, **23**(5):44-53, May 1990.

[26] C. Perkins, *IP Mobility Support for IPv4, revised*, draft-ietf-mobileip-rfc2002-bis-08.txt, Internet Draft, September 2001.

[27] R. Pike, D. Presotto, K. Thompson, and H. Trickey, *Plan 9 from Bell Labs*, Proceedings of the Summer 1990 UKUUG Conference, pp. 1-9, London, July 1990.

[28] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent Checkpointing under Unix*, Proceedings of Usenix Winter 1995 Technical Conference, pp. 213-223, New Orleans, LA, January 1995.

[29] J. Pruyne and M. Livny, *Managing Checkpoints for Parallel Programs*, 2nd Workshop on Job Scheduling Strategies for Parallel Processing (In Conjunction with IPPS '96), Honolulu, Hawaii, April 1996.

[30] X. Qu, J. X. Yu, and R. P. Brent, *A Mobile TCP Socket*, International Conference on Software Engineering (SE '97), San Francisco, CA, November 1997.

[31] R. Rashid and G. Robertson, *Accent: a Communication Oriented Network Operating System Kernel*, Proceedings of the 8th Symposium on Operating System Principles, pp. 64–75, December 1984.

[32] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper, *Virtual Network Computing*, IEEE Internet Computing, **2**(1):33-38, January 1998.

[33] M. Rozier, V. Abrossimov, F. Armand, M. Gien, M. Guillemont, F. Hermann, and C. Kaiser, *Chorus (Overview of the Chorus Distributed Operating System)*, Proceedings of the USENIX Workshop on Micro-Kernels and other Kernel Architectures, Seattle, WA, April 1992.

[34] R. Russell, *Linux 2.4 Packet Filtering HOWTO*, Linux Netfilter Core Team, November 2001. http://netfilter.samba.org/

[35] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, *Optimizing the Migration of Virtual Computers*, Proceedings of the 5th Operating Systems Design and Implementation, Boston, MA, December 2002.

[36] B. K. Schmidt, *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*, Ph.D Thesis, Computer Science Department, Stanford University, August 2000.

[37] D. Senie, *Network Address Translator (NAT)-Friendly Application Design Guidelines*, RFC3235, IETF, January 2002.

[38] A. C. Snoeren and H. Balakrishnan, *An End-to-End Approach to Host Mobility*, Proceedings of 6th International Conference on Mobile Computing and Networking (MobiCom'00), Boston, MA, August 2000.

[39] G. Su and J. Nieh, *Mobile Communication with Virtual Network Address Translation*, Technical Report CUCS-003-02, Department of Computer Science, Columbia University, February 2002.

[40] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound, *Dynamic Updates in the Domain Name System (DNS UPDATE)*, RFC2136, IETF, April 1997.

[41] Y. Zhang and S. Dao, *A "Persistent Connection" Model for Mobile and Distributed Systems*, 4th International Conference on Computer Communications and Networks (ICCCN), Las Vegas, NV, September 1995.

[42] H. Zhong and J. Nieh, *CRAK: Linux Checkpoint/Restart As a Kernel Module*, Technical Report CUCS-014-01, Department of Computer Science, Columbia University, November 2001.

Optimizing the Migration of Virtual Computers

Constantine P. Sapuntzakis Ramesh Chandra Ben Pfaff Jim Chow
Monica S. Lam Mendel Rosenblum
Computer Science Department
Stanford University

{csapuntz, rameshch, blp, jimchow, lam, mendel}@stanford.edu

"Beam the computer up, Scotty!"

Abstract

This paper shows how to quickly move the state of a running computer across a network, including the state in its disks, memory, CPU registers, and I/O devices. We call this state a *capsule*. Capsule state is hardware state, so it includes the entire operating system as well as applications and running processes.

We have chosen to move *x*86 computer states because *x*86 computers are common, cheap, run the software we use, and have tools for migration. Unfortunately, *x*86 capsules can be large, containing hundreds of megabytes of memory and gigabytes of disk data. We have developed techniques to reduce the amount of data sent over the network: copy-on-write disks track just the updates to capsule disks, "ballooning" zeros unused memory, demand paging fetches only needed blocks, and hashing avoids sending blocks that already exist at the remote end. We demonstrate these optimizations in a prototype system that uses VMware GSX Server virtual machine monitor to create and run *x*86 capsules. The system targets networks as slow as 384 kbps.

Our experimental results suggest that efficient capsule migration can improve user mobility and system management. Software updates or installations on a set of machines can be accomplished simply by distributing a capsule with the new changes. Assuming the presence of a prior capsule, the amount of traffic incurred is commensurate with the size of the update or installation package itself. Capsule migration makes it possible for machines to start running an application within 20 minutes on a 384 kbps link, without having to first install the application or even the underlying operating system. Furthermore, users' capsules can be migrated during a commute between home and work in even less time.

1 Introduction

Today's computing environments are hard to maintain and hard to move between machines. These environments encompass much state, including an operating system, installed software applications, a user's individual data and profile, and, if the user is logged in, a set of processes. Most of this state is deeply coupled to the computer hardware. Though a user's data and profile may be mounted from a network file server, the operating system and applications are often installed on storage local to the computer and therefore tied to that computer. Processes are tied even more tightly to the computer; very few systems support process migration. As a result, users cannot move between computers and resume work uninterrupted. System administration is also more difficult. Operating systems and applications are hard to maintain. Machines whose configurations are meant to be the same drift apart as different sets of patches, updates, and installs are applied in different orders.

We chose to investigate whether issues including user mobility and system administration can be addressed by encapsulating the state of computing environments as first-class objects that can be named, moved, and otherwise manipulated. We define a *capsule for a machine architecture* as the data type encapsulating the complete state of a (running) machine, including its operating system, applications, data, and possibly processes. Capsules can be bound to any instance of the architecture and be allowed to resume; similarly, they can be suspended from execution and serialized.

A computer architecture need not be implemented in hardware directly; it can be implemented in software using virtual machine technology[12]. The latter option is particularly attractive because it is easier to extract the state of a virtual computer. Virtual computer states are themselves sometimes referred to as "virtual machines." We introduce the term "capsule" to distinguish the contents of a machine state as a data type from the machinery that can execute machine code. After all, we could bind these machine states to real hardware and not use virtual machines at all.

To run existing software, we chose the standard *x*86 architecture[11, 32] as the platform for our investigation. This architecture runs the majority of operating systems and software programs in use today. In addition, commercial *x*86 virtual machine monitors are available, such as VMware GSX Server (VMM)[28] and Connectix Virtual PC[7], that can run multiple virtual *x*86 machines on the same hardware. They already provide the basic functions of writing out the state of a virtual *x*86 machine, binding the serialized state onto a virtual machine, and resuming execution.

The overall goal of our research project is to explore the design of a capsule-based system architecture, named *the Collective*, and examine its potential to provide user mobility, recovery, and simpler system management. Computers and storage in the Collective system act as caches of capsules. As users travel, the Collective can move their capsules to computers close to them, giving users a consistent environment. Capsules could be moved with users as they commute between home and work. Capsules can be duplicated, distributed to many different machines, and updated like any other data; this can form the basis for administering a group of computers. Finally, capsules can be moved among machines to balance loads or for fail-over.

1.1 Storing and Migrating Capsules

Many challenges must be addressed to realize our goals of the Collective project, but this paper focuses on one simple but crucial one: can we afford the time and space to store, manipulate and migrate *x*86 capsules? *x*86 capsules can be very large. An inactive capsule can contain gigabytes of disk storage, whereas an active capsule can include hundreds of megabytes of memory data, as well as internal machine registers and I/O device states. Copying a gigabyte capsule over a standard 384 kbps DSL link would take 6 hours! Clearly, a straightforward implementation that copies the entire capsule before starting its computation would take too long.

We have developed a number of optimizations that reduce capsules' storage requirements, transfer time and start-up time over a network. These techniques are invisible to the users, and do not require any modifications to the operating system or the applications running inside it. Our techniques target DSL speeds to support capsule migration to and from the home, taking advantage of the availability of similar capsules on local machines.

To speed up the transfer of capsules and reduce the start-up times on slow networks, our system works as follows:

1. Every time we start a capsule, we save all the updates made to disk on a separate disk, using copy-on-write. Thus, a snapshot of an execution can be represented with an incremental cost commensurate with the magnitude of the updates performed.
2. Before a capsule is serialized, we reduce the memory state of the machine by flushing non-essential data to disk. This is done by running a user "balloon" process that acquires memory from the operating system and zeros the data. The remaining subset of memory is transferred to the destination machine and the capsule is started.
3. Instead of sending the entire disk, disk pages are fetched on demand as the capsule runs, taking full advantage of the operating system's ability to tolerate disk fetch latencies.
4. Collision-resistant hashes are used to avoid sending pages of memory or disk data that already exist at the destination. All network traffic is compressed with gzip[8].

We have implemented all the optimizations described in this paper in a basic prototype of our Collective system. Our prototype's platform uses VMware GSX Server 2.0.1 running on Red Hat Linux 7.3 (kernel 2.4.18-10) to execute *x*86 capsules. Users can retrieve their capsules by name, move capsules onto a file system, start capsules on a computer, and save capsules to a file system. We have run both Linux and Windows in our capsules.

Our results show that we can move capsules in 20 minutes or less across 384 kbps DSL, fast enough to move users' capsules between home and work as they commute. Speed improves when an older version of the capsule is available at the destination. For software distribution, we show that our system sends roughly the same amount of data as the software installer package for newly installed software, and often less for upgrades to already installed software. The results suggest that capsule migration offers a new way to use software where machines can start running a new application within a few minutes, with no need to first install the application or even its underlying operating system.

1.2 Paper Organization

Section 2 describes how we use a virtual machine monitor to create and resume capsules. Section 3 motivates the need for optimizations by discussing the intended uses of capsules. Section 4 discusses the optimizations we use to reduce the cost of capsules. In Section 5 we describe some experiments we performed on a prototype of our system. The paper discusses related work in Section 6 and concludes in Section 7.

2 Virtual Machine Monitors

A virtual machine monitor is a layer of software that sits directly on the raw hardware and exports a virtual machine abstraction that imitates the real machine well enough that software developed for the real machine also runs in the virtual machine. We use an *x*86 virtual machine monitor, VMware GSX Server, to generate, serialize, and execute our *x*86 capsules.

Virtual machine monitors have several properties that make them ideal platforms for supporting capsules. The monitor layer encapsulates all of the machine state necessary to run software and mediates all interactions between software and the real hardware. This encapsulation allows the monitor to suspend and disconnect the software and virtual device state from the real hardware and write that machine state to a stream. Similarly, the monitor can also bind a machine state to the real hardware and resume its execution. The monitor requires no cooperation from the software running on the monitor.

Migration is made more difficult by the myriad of hardware device interfaces out there. GSX Server simplifies migration by providing the same device interfaces to the virtual machine regardless of the underlying hardware; virtualization again makes this possible. For example, GSX Server exports a Bus Logic SCSI adapter and AMD Lance Ethernet controller to the virtual machine, independent of the actual interface of disk controller or network adapter. GSX in turn runs on a *host operating system*, currently Linux or Windows, and implements the virtual devices using the host OS's devices and files.

Virtual hard disks are especially powerful. The disks can be backed not just by raw disk devices but by files in the host OS's file system. The file system's abilities to easily name, create, grow, and shrink storage greatly simplify the management of virtual hard disks.

Still, some I/O devices need more than simple conversion routines to work. For example, moving a capsule that is using a virtual network card to communicate over the Internet is not handled by simply remapping the device to use the new computer's network card. The new network card may be on a network that is not able to receive packets for the capsule's IP address. However, since the virtualization layer can interpose on all I/O, it can, transparent to the capsule, tunnel network packets to and from the capsule's old network over a virtual private network (VPN).

3 Usages and Requirements

The Collective system uses serialization and mobility of capsules to provide user mobility, backup, software management and hardware management. We describe each of these applications of capsules and explain their requirements on capsule storage and migration.

3.1 User Mobility

Since capsules are not tied to a particular machine, they can follow users wherever they go. Suppose a user wants to work from home on evenings and weekends. The user has a single active work capsule that migrates between a computer at home and one at work. In this way, the user can resume work exactly where he or she left off, similar to the convenience provided by carrying a laptop. Here, we assume standard home and office workloads, like software engineering, document creation, web browsing, e-mail, and calendar access. The system may not work well with data-intensive applications, such as video editing or database accesses.

To support working from home, our system must work well at DSL or cable speeds. We would like our users to feel that they have instantaneous access to their active environments everywhere. It is possible to start up a capsule without having to entirely transfer it; after all, a user does not need all the data in the capsule immediately. However, we also need to ensure that the capsule is responsive when it comes up. It would frustrate a user to get a screen quickly but to find each keystroke and mouse click processed at glacial speed.

Fortunately, in this scenario, most of the state of a user's active capsule is already present at both home and work, so only the differences in state need to be transferred during migration. Furthermore, since a user can easily initiate the capsule migration before the commute, the user will not notice the migration delay as long as the capsule is immediately available after the commute.

3.2 Backups

Because capsules can be serialized, users and system administrators can save snapshots of their capsules as backups. A user may choose to checkpoint at regular intervals or just before performing dangerous operations. It is prohibitively expensive to write out gigabytes to disk each time a version is saved. Again, we can optimize the storage by only recording the differences between successive versions of a capsule.

3.3 System Management

Capsules can ease the burden of managing software and hardware. System administrators can install and maintain the same set of software on multiple machines by simply creating one (inactive) capsule and distributing it to all the machines. This approach allows the cost of system administration to be amortized over machines running the same configuration.

This approach shares some similarities with the concept of disk imaging, where local disks of new machines are given some standard pre-installed configuration. Disk imaging allows each machine to have only one configuration. On the other hand, our system allows multiple capsules to co-exist on the same machine. This has a few advantages: It allows multiple users with different requirements to use the same machine, e.g. machines in a classroom may contain different capsules for different classes. Also, users can use the same machine to run different capsules for different tasks. They can have a single customized capsule each for personal use, and multiple work capsules which are centrally updated by system administrators. The capsule technique also causes less disruption since old capsules need not be shut down as new capsules get deployed.

Moving the first capsule to a machine over the network can be costly, but may still be faster and less laborious than downloading and installing software from scratch. Moving subsequent capsules to machines that hold other capsules would be faster, if there happen to be similarities between capsules. In particular, updates of capsules naturally share much in common with the original version.

We can also take advantage of the mobility of capsules to simplify hardware resource management. Rather than having the software tied to the hardware, we can select computing hardware based on availability, load, location, and other factors. In tightly connected clusters, this mobility allows for load balancing. Also, migration allows a machine to be taken down without stopping services. On an Internet scale, migration can be used to move applications to servers that are closer to the clients[3].

3.4 Summary

The use of capsules to support user mobility, backup, and system management depends on our ability to both migrate capsules between machines and store them efficiently. It is desirable that our system works well at DSL speed to allow capsules be migrated to and from homes. Furthermore, start-up delays after migration should be minimized while ensuring that the migrated capsules remain responsive.

4 Optimizations

Our optimizations are designed to exploit the property that similar capsules, such as those representing snapshots from the same execution or a series of software upgrades, are expected to be found on machines in a Collective system. Ideally, the cost of storing or transferring a capsule, given a similar version of the capsule, should be proportional to the size of the difference between the two. Also, we observe that the two largest components in a capsule, the memory and the disk, are members of the memory hierarchy in a computer, and as such, many pre-existing management techniques can be leveraged.

Specifically, we have developed the following four optimizations:

1. Reduce the memory state before serialization.
2. Reduce the incremental cost of saving a capsule disk by capturing only the differences.
3. Reduce the start-up time by paging disk data on demand.
4. Decrease the transfer time by not sending data blocks that already exist on both sides.

4.1 Ballooning

Today's computers may contain hundreds of megabytes of memory, which can take a while to transfer on a DSL link. One possibility to reduce the start-up time is to fetch the memory pages as they are needed. However, operating systems are not designed for slow memory accesses; such an approach would render the capsule unresponsive at the beginning. The other possibility is to flush non-essential data out of memory, transfer a smaller working set, and page in the rest of the data as needed.

We observe that clever algorithms that eliminate or page out the less useful data in a system have already been implemented in the OS's virtual memory manager. Instead of modifying the OS, which would require an enormous amount of effort per operating system, we have chosen to use a *gray-box* approach[2] on this problem. We trick the OS into reclaiming physical memory from existing processes by running a *balloon* program that asks the OS for a large number of physical pages. The program then zeros the pages, making them easily compressible. We call this process "ballooning," following the term introduced by Waldspurger[29] in his work on VMware ESX server. While the ESX server uses ballooning to return memory to the monitor, our work uses ballooning to zero out memory for compression.

Ballooning reduces the size of the compressed memory state and thus reduces the start-up time of capsules. This technique works especially well if the memory has many freed pages whose contents are not compressible. There is no reason to transfer such data, and these pages are the first to be cleared by the ballooning process. Discarding pages holding cached data, dirty buffers and active data, however, may have a negative effect. If these pages are immediately used, they will need to be fetched on demand over the network. Thus, even though a capsule may start earlier, the system may be sluggish initially.

We have implemented ballooning in both the Linux and Windows 2000 operating systems. The actual implementation of the ballooning process depends on the OS. Windows 2000 uses a local page replacement algorithm, which imposes a minimum and maximum working set size for each process. To be most effective, the Windows 2000 balloon program must ensure its current working set size is set to this maximum.

Since Linux uses a global page replacement algorithm, with no hard limits on the memory usage of processes, a simple program that allocates and zeros pages is sufficient. However, the Linux balloon program must decide when to stop allocating more memory, since Linux does not define memory usage limits as Windows does. For our tests, the Linux balloon program adopts a simple heuristic that stops memory allocation when free swap space decreases by more than 1MB.

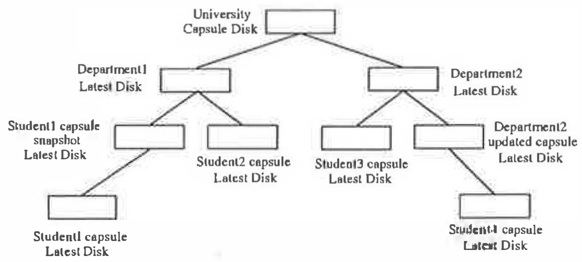
Both ballooning programs explicitly write some zeros to each allocated page so as to stop both OSes from mapping the allocate pages to a single zero copy-on-write page. In addition, both programs hook into the OS's power management support, invoking ballooning whenever the OS receives a suspend request from the VMM.

4.2 Capsule Hierarchies

Capsules in the Collective system are seldom created from scratch, but are mostly derived from other capsules as explained in Section 3. The differences between related capsules are small relative to the total size of the capsules. We can store the disks in these capsules efficiently by creating a hierarchy, where each child capsule could be viewed as inheriting from the parent capsule with the differences in disk state between parent and child captured in a separate copy-on-write (COW) virtual disk.

At the root of the hierarchy is a *root disk*, which is a complete capsule disk. All other nodes represent a COW disk. Each path of COW disks originating from the root in the capsule hierarchy represents a capsule disk; the COW disk at the end of the path for a capsule disk is its *latest disk*. We cannot directly run a capsule whose latest disk is not a leaf of the hierarchy. We must first derive a new child capsule by adding a new child disk to the latest disk and all updates are made to the new disk. Thus, once capsules have children, they become immutable; this property simplifies the caching of capsules.

Figure 1 shows an example of a capsule hierarchy illustrating how it may be used in a university. The root capsule contains all the software available to all students. The various departments in the university may choose to extend the basic capsules with department-specific software. The department administrator can update the department capsule by deriving new child capsules. Students' files are assumed to be stored on networked storage servers; students may use different capsules for different courses; power users are likely to maintain their own private capsule for personal use. Each time a user logs in, he looks up the latest department capsule and derives his own individual capsule. The capsule migrates with the student as he commutes, and is destroyed when he logs out. Note that if a capsule disk is updated, all the derived capsules containing custom installed software have to be ported to the updated capsule. For example, if the University capsule disk is updated, then each department needs to re-create its departmental capsule.


University Capsule Disk
Department1 Latest Disk
Department2 Latest Disk
Student1 capsule snapshot Latest Disk
Student2 capsule Latest Disk
Student3 capsule Latest Disk
Department2 updated capsule Latest Disk
Student1 capsule Latest Disk
Student4 capsule Latest Disk
Figure 1: An example capsule hierarchy.

Capsule hierarchies have several advantages: During the migration of a capsule disk, only COW disks that are not already present at the destination need to be transferred. Capsule hierarchies allow efficient usage of disk space by sharing common data among different capsule disks. This also translates to efficient usage of the buffer cache of the host OS, when multiple capsules sharing COW disks simultaneously execute on the same host. And finally, creating a new capsule using COW disks is much faster than copying entire disks.

Each COW disk is implemented as a bitmap file and a sequence of *extent* files. An extent file is a sequence of blocks of the COW disk, and is at most 2 GB in size (since some file systems such as NFS cannot support larger files). The bitmap file contains one bit for each 16 KB block on the disk, indicating whether the block is present in the COW disk. We use sparse file support of Linux file systems to efficiently store large yet only partially filled disks.

Writes to a capsule disk are performed by writing the data to the latest COW disk and updating its bitmap file. Reads involve searching the latest COW disk and its ancestor disks in turn until the required block is found. Since the root COW disk contains a copy of all the blocks, the search is guaranteed to terminate. Figure 2 shows an example capsule disk and the chain of COW disks that comprise it. Note that the COW disk hierarchy is not visible to the VMM, or to the OS and applications inside the capsule; all of them see a normal flat disk as illustrated in the figure.

The COW disk implementation interfaces with GSX Server through a shim library that sits between GSX Server and the C library. The shim library intercepts GSX Server's I/O requests to disk image files in VMware's "plain disk" format, and redirects them to a local disk server. The plain disk format consists of the raw disk data laid out in a sequence of extent files. The local disk server translates these requests to COW disk requests, and executes the I/O operations against the COW disks.

Each suspend and resume of an active capsule creates a new active capsule, and adds another COW layer to its disks. This could create long COW disk chains. To avoid accumulation of costs in storing the intermediate COW disks, and the cost of looking up bitmaps, we have implemented a *promote* primitive for shortening these chains. We promote a COW disk up one level of the hierarchy by adding to the disk all of its parent's blocks not present in its own. We can delete a capsule by first promoting all its children and then removing its latest disk. We can also apply the promotion operations in succession to convert a COW disk at the bottom of the hierarchy into a root disk.

On a final note, VMware GSX Server also implements a copy-on-write format in addition to its plain disk format. However, we found it necessary to implement our own COW format since VMware's COW format was complex and not conducive to the implementation of the hashing optimization described later in the paper.

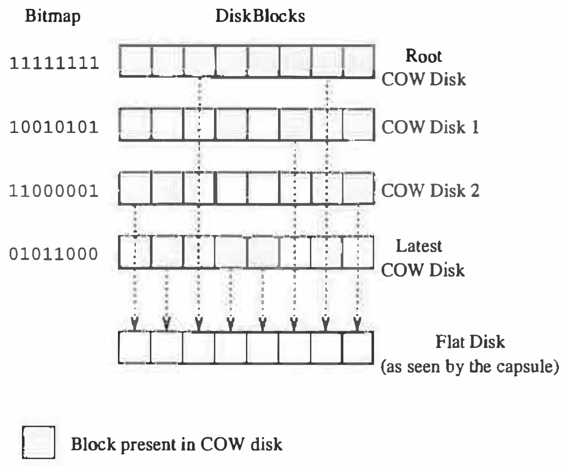


Figure 2: An example capsule disk.

4.3 Demand Paging of Capsule Disks

To reduce the start-up time of a capsule, the COW disks corresponding to a capsule disk are read page-by-page on demand, rather than being pre-fetched. Demand paging is useful because COW disks, especially root disks, could be up to several gigabytes in size and prefetching these large disks could cause an unacceptable start-up delay. Also, during a typical user session, the working set of disk blocks needed is a small fraction of the total blocks on the disk, which makes pre-fetching the whole disk unnecessary. Most OSes have been designed to hide disk latency and hence can tolerate the latency incurred during demand paging the capsule disk.

The implementation of the capsule disk system, including demand paging, is shown in Figure 3. The shim library intercepts all of VMware's accesses to plain disks and forwards them to a disk server on the local machine. The disk server performs a translation from a plain disk access to the corresponding access on the COW disks of the capsule. Each COW disk can either be local or remote. Each remote COW disk has a corresponding local *shadow COW* disk which contains all the locally cached blocks of the remote COW disk.

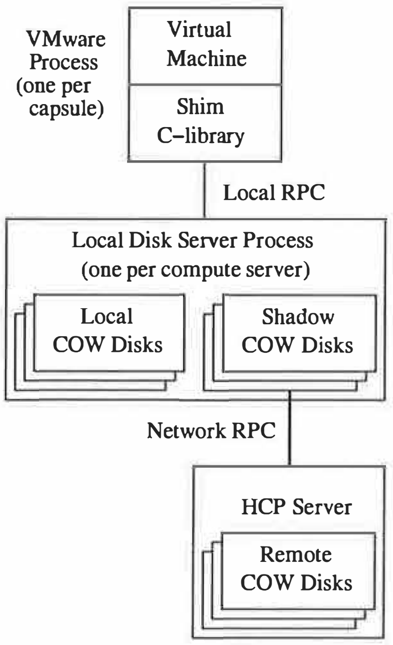


Figure 3: Implementation of capsule disks and demand paging.

Since the latest COW disk is always local, all writes are local. Reads, on the other hand, could either be local or remote. In the case of a remote read, the disk server requests the block from the shadow COW disk. If the block is not cached locally, it is fetched remotely and added to the shadow COW.

Starting a capsule on a machine is done as follows: first, the memory image and all the bitmaps of the COW disks are transferred if they are not available locally. Then the capsule is extended with a new, local latest COW disk. For each remote COW disk, the corresponding shadow COW disk is created if it does not already exist. GSX Server can now be invoked on the capsule. Note that since remote COW disks are immutable, the cached blocks in the shadow COW disks can be re-used for multiple capsules and across suspends and resumes of a single capsule. This is useful since no network traffic is incurred for the cached blocks.

The Collective system uses an LDAP directory to keep track of the hosts on which a COW disk is present. In general, COW disks of a capsule disk could be distributed across many hosts since they were created on different hosts. However, the disks are also uploaded (in the background) to a central storage server for better availability.

4.4 Hash-Based Compression

We use a fourth technique to speed up data transfer over low-bandwidth links. Inspired by the low-bandwidth file system (LBFS[19]) and rsync[27], we decrease transfer time by sending a hash of data blocks instead of the data itself. If the receiver can find data on local storage that hashes to the same value, it copies the data from local storage. Otherwise, the receiver requests the data from the server. We call this technique HCP, for Hashed Copy. The Collective prototype uses HCP for demand paging disks and copying memory and disk images.

We expect to find identical blocks of data between disk images and memories, even across different users' capsules. First, the memory in most systems caches disk blocks. Second, we expect most users in the Collective to migrate between a couple of locations, e.g. home and work. After migrating a couple of times, these locations will contain older memory and disk images, which should contain blocks identical to those in later images, since most users will tend to use the same applications day to day. Finally, most users run code that is distributed in binary form, with most of this binary code copied unmodified into memory when the application runs, and the same binary code (e.g. Microsoft Office or the Netscape web browser) is distributed to millions of people. As a result, we expect to find common blocks even between different users' capsules.

Like LBFS, HCP uses a strong cryptographic hash, SHA-1[1]. The probability that two blocks map to the same 160-bit SHA-1 hash is negligible, less than the error rate of a TCP connection or memory[5]. Also, malicious parties cannot practically come up with data that generates the same hash.

Our HCP algorithm is intended for migrating capsules over low bandwidth links such as DSL. Because HCP involves many disk seeks, its effective throughput is well under 10 Mbps. Hence, it is not intended for high-bandwidth LAN environments where the network is not the bottleneck.

4.4.1 Hash Cache Design

HCP uses a *hash cache* to map hashes to data. Unlike rsync, the cache is persistent; HCP does not need to generate the table by scanning a file or file system on each transfer, saving time.

The cache is implemented using a hash table whose size is fixed at creation. We use the first several bits of the hash key to index into the table. File data is not stored in the table; instead, each entry has a pointer to a file and offset. By not duplicating file data, the cache uses less disk space. Also, the cache can read ahead in the file, priming an in-memory cache with data blocks. Read-ahead improves performance by avoiding additional disk accesses when two files contain runs of similar blocks.

Like LBFS, when the cache reads file data referenced by the table, it always checks that it matches the 20-byte SHA-1 hash provided. This maintains integrity and allows for a couple of performance improvements. First, the cache does not need to be notified of changes to file data; instead, it invalidates table entries when the integrity check fails. Second, it does not need to lock on concurrent cache writes, since corrupted entries do not affect correctness. Finally, the cache stores only the first 8 bytes of the hash in each table entry, allowing us to store more entries.

The hash key indexes into a bucket of entries, currently a memory page in size. On a lookup, the cache does a linear search of the entries in a bucket to check whether one of them matches the hash. On a miss, the cache adds the entry to the bucket, possibly evicting an existing entry. Each entry contains a use count that the cache increments on every hit. When adding an entry to the cache, the hash cache chooses a fraction of the entries at random from the bucket and replaces the entry with the lowest use count; this evicts the least used and hopefully least useful entry of the group. The entries are chosen at random to decrease the chance that the same entry will be overwritten by two parallel threads.

4.4.2 Finding Similar Blocks

For HCP to compress transfers, the sender and receiver must divide both memory and disk images into blocks that are likely to recur. In addition, when demand paging, the operating system running inside the capsule essentially divides the disk image by issuing requests for blocks on the disk. In many systems, the memory page is the unit of disk I/O and memory management, so we chose memory pages as our blocks.

The memory page will often be the largest common unit between different memory images or between memory and disk. Blocks larger than a page would contain two adjacent pages in physical memory; since virtual memory can and does use adjacent physical pages for completely different objects, there is little reason to believe that two adjacent pages in one memory image will be adjacent in another memory image or even on disk.

When copying a memory image, we divide the file into page-sized blocks from the beginning of the image file. For disk images, it is not effective to naively chop up the disk into page-size chunks from the start of the disk; file data on disk is not consistently page aligned. Partitions on *x*86 architecture disks rarely start on a page boundary. Second, at least one common file system, FAT, does not start its file pages at a page offset from the start of the partition. To solve this problem, we parse the partition tables and file system superblocks to discover the alignment of file pages. This information is kept with the disk to ensure we request properly aligned file data pages when copying a disk image.

On a related note, the ext2, FAT, and NT file systems all default to block sizes less than 4 KB when creating smaller partitions; as a result, files may not start on page boundaries. Luckily, the operator can specify a 4 KB or larger block size when creating the file system.

Since HCP hashes at page granularities, it does not deal with insertions and deletions well as they may change every page of a file on disk or memory; despite this, HCP still finds many similar pages.

4.4.3 HCP Protocol

The HCP protocol is very similar to NFS and LBFS. Requests to remote storage are done via remote procedure call (RPC). The server maintains no per-client state at the HCP layer, simplifying error recovery.

Figure 4 illustrates the protocol structure. Time increases down the vertical axis. To begin retrieving a file, an HCP client connects to the appropriate HCP server and retrieves a file handle using the LOOKUP command, as shown in part (a). The client uses READ-HASH to obtain hashes for each block of the file in sequence and looks up all of these hashes in the hash cache. Blocks found via the hash cache are copied into the output file, and no additional request is needed, as shown in part (b). Blocks not cached are read from the server using READ, as in part (c). The client keeps a large number of READ-HASH and READ requests outstanding in an attempt to fill the bandwidth between client and server as effectively as possible.

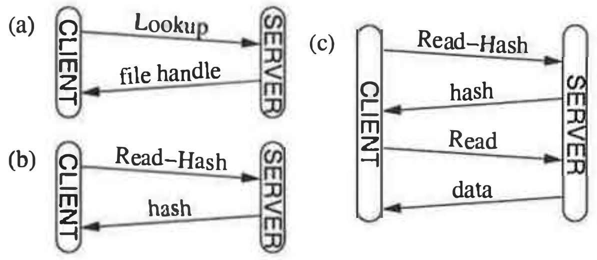


Figure 4: Typical HCP session: (a) session initiation, (b) hash cache hit, (c) hash cache miss.

5 Experimental Results

Our prototype system is based on VMware GSX Server 2.0.1 running on Red Hat Linux 7.3 (kernel 2.4.18-12). Except for the shim library, we wrote the code in Java using Sun's JDK 1.4.0. The experiments ran on 2.4 GHz Pentium 4 machines with 1GB memory.

A separate computer running FreeBSD and the dummynet[23] shaper simulated a 384 kbps symmetric DSL link with 20 ms round-trip delay. We confirmed the setup worked by measuring ping times of 20ms and a TCP data throughput of 360kbps[20]. We checked the correctness of our HCP implementation by ensuring that the hash keys generated are evenly distributed.

We configured the virtual machines to have 256 MB memory and 4 GB local disk. Along with the operating system and applications, the local disk stored user files. In future versions of the system, we expect that user files will reside on a network file system tolerant of low bandwidths.

To evaluate our system, we performed the following four experiments:

1. Evaluated the use of migration to propagate software updates.
2. Evaluated the effectiveness and interaction between ballooning, demand paging, and hash compression.
3. Evaluated the trade-offs between directly using an active capsule versus booting an inactive capsule.
4. Simulated the scenario where users migrate their capsules as they travel between home and work.

5.1 Software Management

Software upgrades are a common system administration task. Consider an environment where a collection of machines maintained to run exactly the same software configuration and users' files are stored on network storage. In a capsule-based system, the administrator can simply distribute an updated capsule to all the machines. In our system, assuming that the machines already have the previous version of the capsule, we only need to send the latest COW disk containing all the changes. Our results show that using HCP to transfer the COW disks reduces the transfer amounts to levels competitive or better than current software install and update techniques. We consider three system administration tasks in the following: upgrading an operating system, installing software packages, and updating software packages.

5.1.1 Operating System Upgrade

Our first experiment is to measure the amount of traffic incurred when updating Red Hat version 7.2 to version 7.3. In this case, the system administrator is likely to start from scratch and create a new root disk, instead of updating version 7.2 and capturing the changes in a COW disk. The installation created a 1.6 GB capsule. Hashing this capsule against a hash cache containing version 7.2 found 30% of the data to be redundant. With gzip, we only need to transfer 25% of the 1.6 GB capsule.

A full operating system upgrade will be a lengthy operation regardless of the method of delivery, due to the large amount of data that must be transferred across the network. Use of capsules may be an advantage for such upgrades because data transfer can take place in the background while the user is using an older version of the capsule being upgraded (or a completely different capsule).

5.1.2 Software Installations and Updates

For this experiment, we installed several packages into a capsule containing Debian GNU/Linux 3.0 and upgraded several packages in a capsule containing Red Hat Linux 7.2. Red Hat was chosen for the latter experiment because out-of-date packages were more readily available.

In each case, we booted up the capsule, logged in as root, ran the Debian apt-get or Red Hat apt-rpm to download and install a new package, configured the software, and saved the capsule as a child of the original one. We migrated the child capsule to another machine that already had the parent cached. To reduce the COW disk size, software packages were downloaded to a temporary disk which we manually removed from the capsule after shutdown.

Figure 5 shows the difference in size between the transfer of the new COW disk using HCP versus the size of the software packages. Figure 5(a) shows installations of some well-known packages; the data point labeled "mega" corresponds to an installation of 492 packages, including the X Window System and TEX. Shown in Figure 5(b) are updates to a number of previously installed applications; the data point labeled "large" corresponds

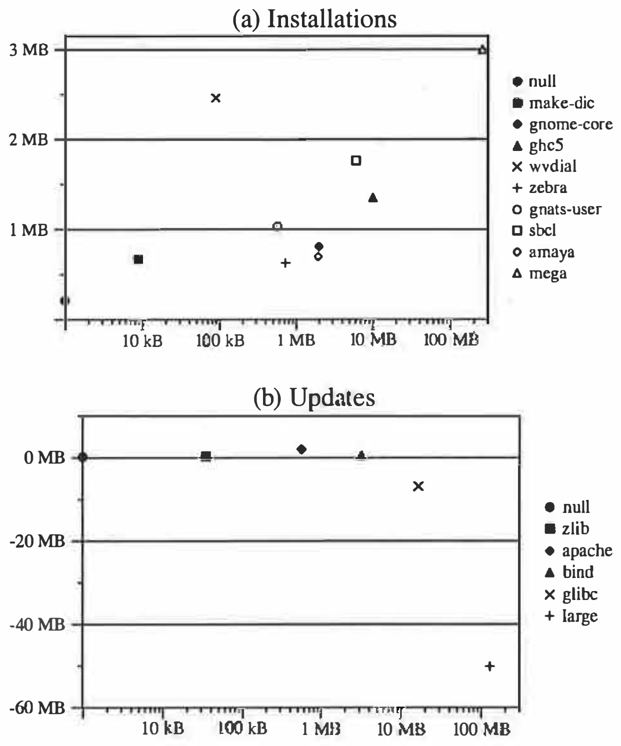


Figure 5: Difference in size between the HCP transfer of the COW disk holding the changes and (a) the installed packages, (b) the update packages.

to an update of 115 packages installed previously and 7 new packages pulled in by the updates. The software updates used were not binary patches; as with an install, they included new versions of all the files in a software package, a customary upgrade method for Debian and Red Hat systems.

For reference, we also include the "null" data point which corresponds to the size of the COW disk created by simply logging in as root and shutting down the capsule without updating any software. This amounts to about 200 KB after HCP and gzip, consisting of i-nodes written due to updated access times, temporary files written at boot, and so on.

As shown in the figure, transfers of small installations and updates are dominated by the installer rewriting two 6 MB text databases of available software. Hashing sometimes saves us from having to send the entire database, but not always, due to insertions that change all the pages. The different results for make-dic and wvdial illustrate this effect. On larger installs, the cost of transferring the disk via HCP is near that of the original package; the overhead of the installer database is bounded by a constant and gzip does a good job of compressing the data. For larger updates, HCP sent less data than the packages because many of these updates contained only minor changes from previous versions (such as security patches and bug fixes), so that hashing found similarities to older, already installed packages. In our experiment, for updates over 10 MB, the savings amount to about 40% in each case.

The results show that distributing COW disks via HCP is a reasonable alternative to current software install and update techniques. Package installations and upgrades incur a relatively low fixed cost, so further benefits can be gained by batching smaller installs. In the case of updates, HCP can exploit similarities between the new and old packages to decrease the amount of data transferred. The convenience of a less tedious and error-prone update method is another advantage.

5.2 Migration of Active Capsules

To show how capsules support user mobility, we performed two sets of experiments, the first on a Windows 2000 capsule, the second on a Linux capsule.

First, we simulated the workload of a knowledge worker with a set of GUI-intensive applications on the Windows 2000 operating system. We used Rational Visual Test software to record user activities and generate scripts that can be played back repeatedly under different test conditions. We started a number of common applications, including Microsoft Word, Excel, and PowerPoint, plus Forte, a Java programming environment, loaded up some large documents and Java sources, saved the active capsule, migrated it to another machine, and proceeded to use each of the four applications.

On Linux, we tested migration with less-interactive and more CPU- and I/O-bound jobs. We chose three applications: processor simulation with `smttls`, Linux kernel compilations with GCC, and web serving with Apache. We imagine that it would be useful to migrate large processor simulations to machines that might have become idle, or migrate fully configured webservers dynamically to adjust to current demand. For each experiment, we performed a task, migrated the capsule, then repeated the same task.

To evaluate the contributions of each of our optimizations, we ran each experiment twelve times. The experimental results are shown in Figure 6. We experimented with two network speeds, 384 kbps DSL and switched 100 Mbps Ethernet. For each speed, we compared the performance obtained with and without the use of ballooning. We also varied the hashing scheme: the experiments were run with no hashing, with hashing starting from an empty hash cache, and with hashing starting from a hash cache primed with the contents of the capsule disk. Each run has two measurements: "migration," the data or time to start the capsule, and "execution," the data or time it took to execute the task once started.

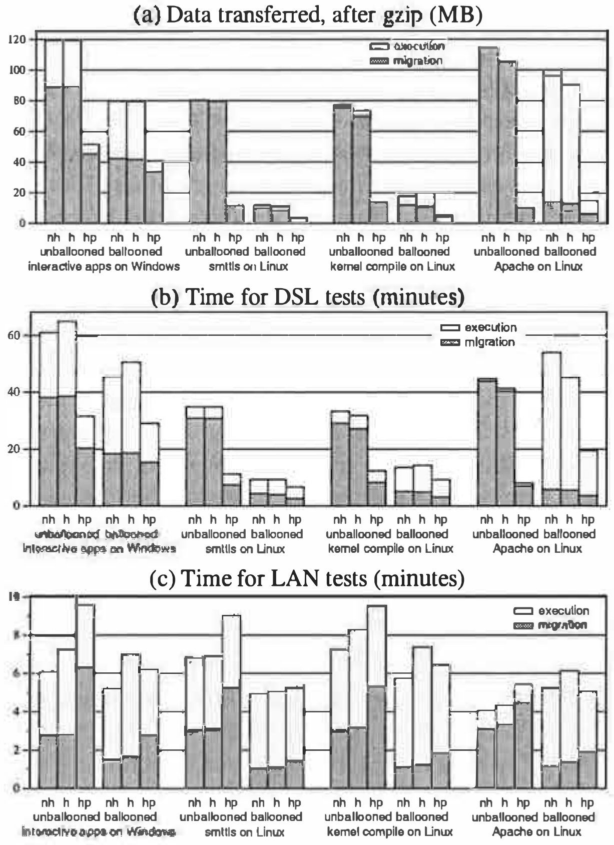


Figure 6: Migration experiments. Data transferred for remote activations and executions are shown after gzip in (a). Time to activate and run the experiments are shown for 384 kbps DSL in (b) and 100 Mbps switched Ethernet in (c). Labels "nh", "h", and "hp" denote no hashing, hashing with an empty hash cache, and hashing with a primed cache, respectively.

Figure 6(a) shows the amounts of data transferred over the network during each migration and execution after gzip. These amounts are independent of the network speed assumed in the experiment. The memory image is transferred during the migration step, and disk data are transferred on demand during the execution step. Gzip by itself compresses the 256 MB of memory data to 75–115 MB. Except for the Windows interactive benchmark, none of the applications incurs uncached disk accesses during unballooned execution.

Hashing against an empty cache has little effect because it can only find similarities within the data being transferred. Our results show that either hashing against a primed disk or ballooning alone can greatly reduce the amount of memory data transferred to 10–45 MB. By finding similarities between the old and new capsules, primed hashing reduces the amount of data transferred both during migration and execution. While ballooning reduces the amount of memory data that needs to be transferred, it does so with the possible expense of increasing the data transferred during the execution. Its effectiveness in reducing the total amount of data transferred is application-dependent; all but Apache, which has a large working set, benefit tremendously from ballooning. Combining ballooning with primed hashing generally results in the least amount of data transferred.

The timing results obtained on a 384 kbps DSL link, shown in Figure 6(b), follow the same pattern found in Figure 6(a). The execution takes longer proportionally because it involves computation and not just data transfer. With no optimization, it takes 29–44 minutes just to transfer the memory image over before the capsule can start executing. Hashing with priming reduces the start-up to less than 20 minutes in the Windows interactive experiment, and less than 6 minutes in all the other applications. Ballooning also reduces the start-up time further to 3–16 minutes. Again, combining both ballooning and priming yields the best result in most cases. As the one exception here, Apache demonstrates that ballooning applications with a large working set can slow them down significantly.

Hashing is designed as an optimization for slow network connections; on a fast network, hashing can only slow the transfer as a result of its computational overhead. Figure 6(c) shows this effect. Hashing against a primed cache is even worse because of the additional verification performed to ensure that the blocks on the destination machine match the hash. This figure shows that it takes only about 3 minutes to transfer an unballooned image, and less than 2 minutes ballooned. Again, except for Apache which experiences a slight slowdown, ballooning decreases both the start-up time as well as the overall time.

The Windows experiment has two parts, an interactive part using Word, Excel, and PowerPoint on a number of large documents, followed by compiling a source file and building a Java archive (JAR) file in Forte. The former takes a user about 5 minutes to complete and the latter takes about 45 seconds when running locally using VMware. In our test, Visual Test plays back the keystrokes and mouse clicks as quickly as possible. Over a LAN with primed hashing, the interactive part takes only 1.3 minutes to complete and Forte takes 1.8 minutes. Over DSL with primed hashing, the interactive part takes 4.4 minutes and Forte takes 7 minutes. On both the DSL and LAN, the user sees an adequate interactive response. Forte is slower on DSL because it performs many small reads and writes. The reads are synchronous and sensitive to the slow DSL link. Also, the first write to a 16 KB COW block will incur a read of the block unless the write fills the block, which is rarely the case.

The processor simulation, kernel compile, and Apache tasks take about 3, 4, and 1 minutes, respectively, to execute when running under VMware locally. Without ballooning, these applications run mainly from memory, so remote execution on either LAN or DSL is no slower than local execution. Ballooning, however, can increase run time, especially in the case of Apache.

Our results show that active capsules can be migrated efficiently to support user mobility. For users with high-speed connectivity, such as students living in a university dormitory, memory images can be transferred without ballooning or hashing in just a few minutes. For users with DSL links, there are two separate scenarios. In the case of a commute between work and home, it is likely that an earlier capsule can be found at the destination, so that hashing can be used to migrate an unballooned memory image. However, to use a foreign capsule, ballooning is helpful to reduce the start-up time of many applications.

5.3 Active Versus Inactive Capsules

The use of capsules makes it possible for a machine in a Collective system to run an application without first having to install the application or even the operating system on which the application runs. It is also possible for a user to continue the execution of an active capsule on a different machine without having first to boot up the machine, log on, and run the application.

We ran experiments to compare these two modes of operation. These experiments involved browsing a webpage local to the capsule using Mozilla running on Linux. From the experiment results, we see that both active and inactive capsules are useful in different scenarios and that using capsules is easier and takes less time than installing the required software on the machine.

The different scenarios we considered are:

1. We mounted the inactive capsule file using NFS over the DSL link. We booted the inactive capsule, ran Mozilla, and browsed a local webpage. The results for this test are shown in Figure 7 with the label NFS.

2. We used demand paging to boot the inactive capsule and ran Mozilla to browse the local webpage. We considered three alternatives: the machine had not executed a similar capsule before and therefore had an empty hash cache, the machine had not executed the capsule before but the hash cache was primed with the disk state of the capsule, and the machine had executed the same capsule before and hence the capsule's shadow disk had the required blocks locally cached. The results are shown in Figure 7 un-

der the labels boot, boot-p, and boot2 respectively.

3. We activated an active remote capsule that was already running a browser. We ran it with and without ballooning, and with and without priming the hash cache with the inactive capsule disk. The results are shown in the figure under the labels active, active-b, active-p, and active-bp.

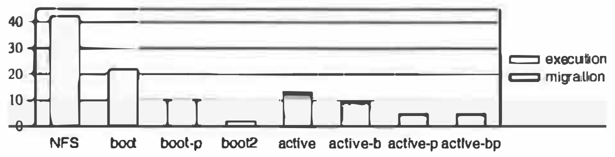


Figure 7: Times for activating a browser capsule (in minutes). The capsules are NFS, booted with an NFS-mounted disk; boot, a remote capsule booted with demand paging and unprimed database; boot2, the same remote capsule booted a second time; and active, migration of a capsule with a running browser. Suffix "b" indicates that ballooning was done and suffix "p" indicates that a primed database was used.

The bars in Figure 7 show the time taken while performing the test. The times are split into execution and migration times. As expected, the four inactive capsules in the first two scenarios have negligible migration times, while execution times are negligible for the four active capsules in the last scenario. When comparing the different scenarios we consider the total times as a sum of migration time and execution time.

In this experiment, demand paging, even with an empty hash cache, performed much better than NFS. Demand paging brought down the total time from about 42 minutes for NFS to about 21 minutes. When the cache was warmed up with a similar capsule, the total time for the inactive capsule dropped to about 10 minutes. When the inactive capsule was activated again with the required blocks locally cached in the capsule's shadow disk, it took only 1.8 minutes, comparable to boot of a local capsule with VMware. Using an active capsule with no ballooning or priming required about 12 minutes. Ballooning the active capsule brought the time down to about 10 minutes, and priming the hash cache brought it down further to about 4 minutes, comparable to the time taken to boot a local machine and bring up Mozilla. These times are much less than the time taken to install the required software on the machine.

These results suggest that: (a) if a user has previously used the inactive capsule, then the user should boot that capsule up and use it, (b) otherwise, if the user has previously used a similar capsule, the user should use an active capsule, and (c) otherwise, if executing the capsule for the first time, the user should use an active ballooned capsule.

5.4 Capsule Snapshots

We simulate the migration of a user between work and home machines using a series of snapshots based on the Business Winstone 2001 benchmark. These simulation experiments show that migration can be achieved within a typical user's commute time.

The Winstone benchmark exercises ten popular applications: Access, Excel, FrontPage, PowerPoint, Word, Microsoft Project, Lotus Notes, WinZip, Norton AntiVirus, and Netscape Communicator. The benchmark replays user interaction as fast as possible, so the resulting user session represents a time-compressed sequence of user input events, producing large amounts of stress on the computer in a short time.

To produce our Winstone snapshots, we ran one iteration of the Winstone test suite, taking complete images of the machine state every minute during its execution. We took twelve snapshots, starting three minutes into the execution of the benchmark. Winstone spends roughly the first three minutes of its execution copying the application programs and data it plans to use and begins the actual workload only after this copying finishes. The snapshots were taken after invoking the balloon process to reduce the user's memory state.

To simulate the effect of a user using a machine alternately at work and home, we measured the transfer of snapshot to a machine that already held all the previous snapshots. Figure 8 shows the amount of data transferred for both the disk and memory images of snapshot 2 through 12. It includes the amount of data transferred with and without hashing, and with and without gzip.

The amount of data in the COW disk of each snapshot varied depending on the amount of disk traffic that Winstone generated during that snapshot execution. The large size of the snapshot 2 COW disk is due to Winstone copying a good deal of data at the beginning of the benchmark. The size of the COW disks of all the other snapshots range from 2 to 22 MB after gzip, and can be transferred over completely under about 8 minutes. Whereas hashing along with gzip compresses the COW disks to about 10–30% of their raw size, it compresses the memory images to about 2–6% of their raw size. The latter reduction is due to the effect of the ballooning process writing zero pages in memory. The sizes of ballooned and compressed memory images are fairly constant across all the snapshots. The memory images require a transfer of only 6–17 MB of data, which takes no more than about 6 minutes on a DSL link. The results suggest that the time needed to transfer a new memory image, and even the capsule disk in most cases, is well within a typical user's commute time.

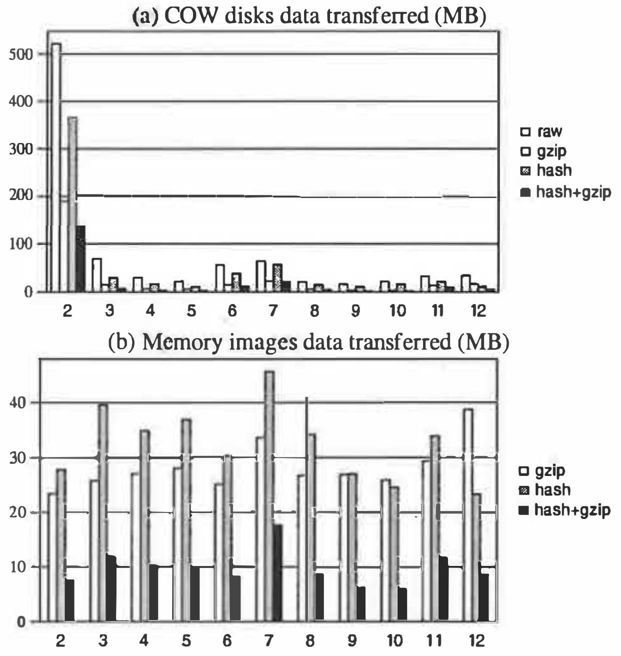


Figure 8: Snapshots from the Winstone benchmark showing (a) disks and (b) memory images transferred. Raw sizes not shown in (b) are constant at about 256 MB.

6 Related Work

Much work was done in the 1970s on virtual machines at the hardware level[9] and interest has recently revived with the Disco[4] and Denali[30] projects and VMware GSX Server[28]. Earlier work demonstrated the isolation, performance, and economic properties of virtual machines. Chen and Noble suggested using hardware-level virtual machines for user mobility[6]. Kozuch and Satyanarayanan independently came up with the idea of using VMware's *x*86 VMMs to achieve mobility[15].

Others have also looked at distributing disk images for managing large groups of machines. Work by Rauch et al. on partition repositories explored maintaining clusters of machines by distributing raw disk images from a central repository[22]. Rauch's focus is on reducing the size of the repository; ours is on reducing time spent sending disk images over a WAN. Their system, like ours, reduces the size of successive images by storing only the differences between revisions. They also use hashes to detect duplicate blocks and store only one copy of each block. Emulab[31], Cluster-on-Demand[18], and others, are also distributing disk images to help maintain groups of computers.

The term *capsule* was introduced earlier by one of the authors and Schmidt[24]. In that work, capsules were implemented in the Solaris operating system and only groups of Solaris processes could be migrated.

Other work has looked at migration and checkpointing at process and object granularities. Systems working at process level include V[26], Condor[16], libckpt[21], and CoCheck[25]. Object-level systems include Legion[10], Emerald[14], and Rover[13].

LBFS[19] provided inspiration for HCP and the hash cache. Whereas LBFS splits blocks based on a fingerprint function, HCP hashes page-aligned pages to improve performance on memory and disk images. Manber's SIF[17] uses content-based fingerprinting of files to summarize and identify similar files.

7 Conclusions

In this paper, we have shown a system that moves a computer's state over a slow DSL link in minutes rather than hours. On a 384kbps DSL link, capsules in our experiments move in at most 20 minutes and often much less.

We examined four optimization techniques. By using copy-on-write (COW) disks to capture the updates to disks, the amount of state transferred to update a capsule is proportional to the modifications made in the capsule. Although COW disks created by installing software can be large, they are not much larger than the installer and more convenient for managing large numbers of machines. Demand paging fetches only the portion of the capsule disk requested by the user's tasks. "Ballooning" removes non-essential data from the memory, thus decreasing the time to transfer the memory image. Together with demand paging, ballooning leads to fast loading of new capsules. Hashing exploits similarities between related capsules to speed up the data transfer on slow networks. Hashing is especially useful for compressing memory images on user commutes and disk images on software updates.

Hopefully, future systems can take advantage of our techniques for fast capsule migration to make computers easier to use and maintain.

8 Acknowledgments

This research is supported in part by National Science Foundation under Grant No. 0121481 and Stanford Graduate Fellowships. We thank Charles Orgish for discussions on system management, James Norris for working on an earlier prototype, our shepherd Jay Lepreau, Mike Hilber, and David Brumley for helpful comments.

References

[1] FIPS 180-1. Announcement of weakness in the secure hash standard. Technical report, National Institute of Standards and Technology (NIST), April 1994.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–59, October 2001.

[3] A. A. Awadallah and M. Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Seventh International Workshop on Web Content Caching and Distribution*, August 2002.

[4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[5] F. Chabaud and A. Joux. Differential collisions in SHA-0. In *Proceedings of CRYPTO '98, 18th Annual International Cryptology Conference*, pages 56–71, August 1998.

[6] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the 8th IEEE Workshop on Hot Topics on Operating Systems*, May 2001.

[7] http://www.connectix.com/.

[8] P. Deutsch. Zlib compressed data format specification version 3.3, May 1996.

[9] R. P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, June 1974.

[10] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Legion: An operating system for wide-area computing. Technical Report CS-99-12, Dept. of Computer Science, University of Virginia, March 1999.

[11] IA-32 Intel architecture software developer's manual volumes 1-3. http://developer.intel.com/design/pentium4/manuals/.

[12] *IBM Virtual Machine/370 Planning Guide.* IBM Corporation, 1972.

[13] A. Joseph, J. Tauber, and M. Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, March 1997.

[14] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transaction on Computer Systems*, 6(1):109–133, February 1988.

[15] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 40–46, June 2002.

[16] M. Litzkow, M. Livny, and M. Mutka. Condor – a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[17] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, 17–21 1994.

[18] J. Moore and J. Chase. Cluster on demand. Technical report, Duke University, May 2002.

[19] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, October 2001.

[20] M. Muuss. The story of T-TCP.

[21] J. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the USENIX Winter 1995 Technical Conference*, pages 213–224, January 1995.

[22] F. Rauch, C. Kurmann, and T. Stricker. Partition repositories for partition cloning—OS independent software maintenance in large clusters of PCs. In *Proceedings of the IEEE International Conference on Cluster Computing 2000*, pages 233–242, 2000.

[23] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, Jan. 1997.

[24] B. K. Schmidt. *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*. PhD thesis, Computer Science Department, Stanford University, August 2000.

[25] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531, April 1996.

[26] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proc. 10th Symposium on Operating Systems Principles*, pages 10–12, December 1985.

[27] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, April 2000.

[28] "GSX server", white paper. http://www.vmware.com/pdf/gsx_whitepaper.pdf.

[29] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[30] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical report, University of Washington, February 2001.

[31] B. White et al. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[32] Wintel architecture specifications. http://www.microsoft.com/hwdev/specs/.

Luna: a Flexible Java Protection System

Chris Hawblitzel, Dartmouth College

Thorsten von Eicken, Expertcity, Inc.

Abstract

Extensible Java systems face a difficult trade-off between sharing and protection. On one hand, Java's ability to run different protection domains in a single virtual machine enables domains to share data easily and communicate without address space switches. On the other hand, unrestricted sharing blurs the boundaries between protection domains, making it difficult to terminate domains and enforce restrictions on resource usage. Existing solutions to these problems restrict sharing in an ad-hoc fashion, ruling out many desirable programming styles.

This paper presents an extension to Java's type system that systematically addresses the issues of data sharing, revocation, thread control, and resource control. Multiple *tasks* running in a single virtual machines share data using special *remote pointers*, which have different types from local pointers. The distinction between local and remote pointers allows the Java run-time system to mediate the communication between tasks without slowing down operations on ordinary pointers. The extensions to Java are implemented by a system called Luna, based on the Guavac and Marmot compilers, extended with special optimizations to support both fast inter-task communication and dynamic access control. The paper describes two applications written in Luna: a simple extensible web server, and an extension of the Squid web cache to support dynamic content generation.

1. Introduction

Traditional operating systems such as Unix, Windows NT, and more recent microkernels use virtual memory to protect different programs from one another and to divide a computer's finite resources among the programs. In these systems, the *process* or *task* is the central unit of protection and resource control, and cleanly encapsulates each program's memory usage and processor usage. In recent years, language-based protection has become a viable alternative to virtual memory based protection for many types of extensible applications. For instance, the Java programming language is now used to extend browsers with applets, servers with servlets [Java], active network routers with new protocols [WGT98], databases with customizations [GMS+98], and agent systems with mobile agents [Gen]. In these systems, the language subsumes the role of a traditional operating system, and is responsible for protecting programs from one another.

Language-based protection relies on the safety of a programming language's type system to restrict the operations that a program is allowed to perform. The language provides a set of types (integers, functions, and records, for instance), and operations that can manipulate instances of different types. Some operations make sense for some types but not others. For instance, a Java program can invoke a method of an object, but it cannot perform a method invocation on an integer.

Type-safe languages implement capability based access control very naturally: a pointer (also called a *reference* in Java jargon) cannot be forged, and can therefore serve as a capability. Languages typically provide additional access control mechanisms, such as Java's `private`, default, `protected`, `public`, and `final` access qualifiers that specify which code has access to which fields and methods of an object. Wallach et al [WBD+97] discusses Java access control mechanisms in detail.

Safe languages hold many potential advantages over virtual memory based systems: fine-grained sharing, no expensive address space switches, natural capability-based access control, abstract datatype enforcement, and code portability. Unfortunately, this list of language protection advantages is matched by an equally long list of drawbacks. The performance of safe languages tends to lag behind lower level languages like C, typical language-based systems support only one language, and relying on a language for protection requires trusting that the language's compiler (the just-in-time compiler in Java's case) and run-time system are written correctly. Other research has made great strides in attacking these problems ([BSP+95], [HLP98], [MWC+98], [NL98], [Sha97]). This paper focuses on a different set of problems, arising from the lack of OS features and structures in current language-based systems.

For example, early Java systems suffered from unexpected interactions between data, threads, and code. Consider Java's `Thread.stop` and `ThreadGroup.stop` methods, which asynchronously terminate a thread or group of threads. These would seem ideal for stopping a runaway applet that is consuming all of a browser's CPU time and memory. However, closer inspection reveals several difficult problems:

- The wrong code gets interrupted: in Java, a call from an applet to the system is just a function call, so that the system code runs in the applet's thread. The applet can kill or suspend the thread while the system code is running, possibly leaving the system in an inconsistent or deadlocked state.
- Malicious code eludes termination: terminating a malicious program's threads does not terminate the program's code, which may still exist in overridden methods of objects that the malicious program created and passed to other protection domains. If one of these other domains invokes the rogue object methods, the malicious code is revived. Java mitigates this problem by making key datatypes *final* (e.g. `String`, array types) or *primitive* (e.g. `int`, `float`) so that they cannot contain overridden methods, but a purer object-oriented language would have more difficulties.
- Damaged objects violate abstract datatype integrity: under Java's synchronization mechanisms, a thread may enter an object's method, acquire a lock, start an atomic operation, and then get terminated in the middle of the operation, releasing the lock and leaving the object in a "damaged" state that violates its intended abstraction.
- Resources aren't reclaimed: when a traditional operating system shuts down a process, both the process's threads and memory are shut down. Java's termination is weaker, stopping threads but not necessarily reclaiming other resources.

Sun deprecated the `Thread.stop` and `ThreadGroup.stop` methods, leaving no officially sanctioned way to stop an applet [Javb]. The Java Community Process is currently considering a new API for isolating protection domains, which would prevent the direct sharing of objects and threads between domains. In this case, it is much easier to terminate an applet and reclaim its resources. While a better isolation mechanism is a clear step forward for many applications, these applications still require communication between programs. The design of the communication mechanism is the key to obtaining the promised advantages of language-based protection, such as fine-grained sharing.

The rest of this paper discusses existing research on combining isolation and communication, and then attempts to synthesize desirable features of different approaches into a single model, where *tasks* implement isolation and *remote pointers* enable communication between tasks. The paper describes a type system for remote pointers, and then describes how tasks and remote pointers are implemented and optimized in an extension to Java that we call *Luna*. Luna is implemented as a source-level extension to the Guavac source-to-bytecode compiler, which produces special Luna bytecode that runs on an extension to the Marmot, a sophisticated optimizing Java virtual machine [FKR+99]. The extensions to the source language are small, while the extensions to Marmot's run-time system are more involved. Note that tasks and remote pointers would apply to languages besides Java; an earlier paper [HvE99] describes the application of these ideas both in a C-like procedural language with modules and as a formal extension of the typed lambda calculus for which we proved soundness.

2. Balancing sharing and isolation: existing approaches

The two goals of sharing and isolation conflict with each other, and building a system to support both requires compromises. For example, DrScheme/MrEd [FFK+99] restricts communication to a hierarchical parent-to-child pattern, providing no support for peer-to-peer sharing (e.g. applet-to-applet or agent-to-agent sharing), in order to allow a parent to cleanly terminate a child. DrScheme's approach also leaves much of the responsibility for clean termination to the programmer. For example, the parent is expected to explicitly relinquish pointers to objects in a child domain in order to avoid damaged objects and ensure memory reclamation; MrEd's "custodians" do not do this automatically.

At the opposite extreme, the J-Kernel [HCC+98] allows communication between arbitrary protection domains (even mutually suspicious domains), but only allows the domains to share special "capability" objects through which they can perform remote method invocations. The system ensures that these capability objects are revocable, so that when a protection domain is shut down, the capabilities that it exported are revoked and the domain's memory is reclaimed. Bryce et al [BR00] describe a similar "object space" model that makes it easier to share indirect references to objects, but still does not allow direct sharing of arrays or object fields. The Java Community Process is also considering a mechanism to let isolated programs pass data by copy. Unfortunately, these heavyweight approaches abandon many of the advantages of lightweight protected sharing, and rule out programming styles based on shared memory.

The J-Kernel and object space approaches contrast with KaffeOS [BHL00], which makes sharing of byte arrays and primitive data fields easy. KaffeOS is oriented towards a shared memory style of programming, at the expense of more object-oriented programming styles based on method invocation. For example, KaffeOS "shared heaps" cannot contain objects whose methods may be overridden arbitrarily, because such sharing would let one domain's thread call another domain's code (causing the problems with domain termination described in the previous section). Java is very object-oriented, though, most Java classes have overrideable methods. This means, for example, that KaffeOS must prohibit the sharing of an object containing a field of type `Object`, because this field might hold an object that overrides one of `Object`'s methods, such as `equals` or `hashCode`. A purer object-oriented language, where all methods may be overridden, would be even worse.

3. Remote pointers and the task model

All the systems in the previous section strive to draw clear boundaries between different domains. This common goal motivates DrScheme's parent-to-child communication model, the J-Kernel's restriction on object sharing, and KaffeOS's restriction on shared object types.

We propose expressing the domain boundaries explicitly in Luna's type system, in order to build a system that supports object-oriented remote method invocations, shared memory, and arbitrary peer-to-peer communication in a natural way. In our approach, a single virtual machine will contain many *tasks*, each with its own objects, threads, and code. Inter-task communication is organized around the concept of a revocable *remote pointer*, which is built into Luna's type system. Pointers from one task to an object in another task have a special type indicated by a tilde (e.g. "`String~`", "`Hashtable~`"). These remote pointers are dynamically revocable, so that when a task is shut down, pointers into the task are revoked, and the task's objects, code and threads are safely deallocated. This allows fine-grained sharing between arbitrary tasks to coexist with clean termination, effective thread and resource control, and powerful optimization, as discussed in the following subsections.

Resource control

When objects are shared at a fine granularity, which programs should be charged for the objects and how can memory usage be tracked? One solution is to charge for all the objects reachable from a program's roots. While this may work in traditional operating systems with fixed sized shared memory buffers of "raw data", it is dangerous for abstract data types and pointer based data structures. Program A can give program B a single object with some private field pointing to all of A's private data, and program B gets charged for all of A's data. Fine-grained sharing looks less pleasant when any shared object can act as a resource Trojan horse. Luna offers a simple alternative: a task pays only for those objects that it explicitly allocates with the Java `new` operator; it is not charged for objects allocated by other tasks.

Revocation

In ordinary Java, pointers can serve as capabilities, but once a program is given a pointer to an object, that pointer cannot be revoked. Why would anyone want to revoke a capability? Revocation assists termination: Luna revokes all remote pointers into a task when the task is terminated. This neatly solves the damaged object problem: a dead task may contain damaged objects, but revocation ensures that these objects are not accessible from other tasks. Revocation also helps to implement the principle of least privilege, since it is better to give someone access to something for only the necessary duration and then revoke the access than to give them access forever. Revocation accommodates changing preferences over time. An agent once considered trustworthy may abuse the trust and necessitate revoking its privileges. At a lower level, revocation is a good way to give someone temporary, fast access to a resource, such as idle network buffer space or a rectangle in video memory. In addition to these device-specific examples, revocation is also used in general purpose operating system mechanisms: for instance, FBufs [DP93] dynamically revoke write access to buffers when data is transferred between protection domains.

Revocation has been a historical problem for capability systems [WLH81], but it is particularly difficult at the language level. While adding a level of indirection to every capability may be a way to implement revocation in an operating system with coarse-grained capabilities [Red74], adding a level of indirection to every Java object is undesirable. Luna's static distinction between local and remote pointers ensures that only pointers shared between tasks incur the overhead of supporting revocation (as described below, the high cost of locks on modern processors makes this overhead a serious concern).

Controlling threads

If a call from one program to another is just a function call, as it is in standard Java, the caller and callee programs share the same thread. Who is allowed to terminate or suspend the shared thread? In current Java browsers, an applet can call the browser and then kill the thread while the browser is in the middle of a sensitive operation. Luna, on the other hand, switches threads during a method invocation on a remote pointer, so that the caller and callee each have their own threads, which is similar to IPC and RPC mechanisms in traditional operating systems. Luna's remote pointer types statically signal which method invocations are on local objects (needing no thread switch) and which are *cross-task* method invocations on remote objects.

Reasoning about the system structure

Analyzing the security of a system requires analyzing the communication patterns between programs. Limited, well-defined communication channels make this easy; unconstrained fine-grained sharing makes a mess. Luna's remote pointer types statically mark the boundaries between tasks, and thus form a static blueprint of the system's cross-task communication patterns.

Optimization

While dynamic loading is a useful tool for system extensibility, many optimization techniques rely on global information (such as "this method is never overridden and may therefore be inlined") that may be invalidated as code is dynamically loaded. One way to reconcile these *whole-program optimizations* with dynamic loading is to undo optimizations as necessary when new code is loaded. Suppose a browser uses the Java class `Vector`, but never overrides the methods of this class. In this situation, the browser may inline method invocations on objects of type `Vector`. If a newly loaded applet introduces a class (say, `SortedVector`) that overrides these methods, however, the browser's code must be dynamically recompiled to remove the inlining, because the applet might pass a `SortedVector` object to the browser, and the browser might invoke one of the object's overridden methods. Dynamic recompilation is complicated to implement, requiring close cooperation between the compiler and run-time system. Moreover, in a language-based protection system, where multiple programs are loaded into a single environment, it penalizes one program for the actions of another program. Why should a browser have to undo its internal optimizations because of code contained in a dynamically loaded applet? It is difficult to predict the performance a program when its optimization depends on other programs' code.

Luna solves this problem by making the task the fundamental unit of loading, instead of loading classes individually. This enables "whole-task optimizations" of operations on local pointers, so that Luna is able to exploit Marmot's inlining and static method binding. In the example above, the browser would continue to make inlined calls to its own `Vector` objects, because a local pointer of type `Vector` cannot point to the applet task's `SortedVector` objects. The only Marmot whole-program optimization that Luna does not implement is object stack allocation, which adds extra method table entries to some classes (Luna must ensure that method tables and field layouts are consistent across tasks, to support operations on remote pointers).

4. Remote pointers

In order to preserve the advantages of safe language protection (fine-grained sharing, low cross-task call overheads, simple capability-based access control, and enforcement of abstract data types), remote pointers support the same operations as local pointers: field/array element access, method invocation, synchronization, equality testing, casting, and instanceof testing, although most of these operations have different semantics and performance for remote pointers. As Figure 1 indicates, there is one remote pointer type for each class/interface type and array type. For convenience, we will refer to the objects pointed to by local and remote pointers as "local objects" and "remote objects" respectively.

```
Type = PrimitiveType
     | ReferenceType
     | ReferenceType~
PrimitiveType = boolean | byte
     | short | int | long | char
     | float | double
ReferenceType = ClassType
                | InterfaceType
                | Type[]
```

Figure 1: Luna's type system

The key difference between remote pointers and local pointers is revocation. Luna's task model requires that remote pointers into a task be revoked when the task is killed, but the previous section argues that revocation is also useful at a finer granularity. To realize these uses, Luna gives the programmer a special handle with which to control access to remote pointers. This handle is a Java object called a *permit*. A permit is allocated in an unrevoked state, and can later be revoked:

```
public class Permit {
    public Permit();
    public void revoke();
    ...
}
```

A remote pointer is implemented as a two-word value that consists of a local pointer paired with a permit. The @ operator converts a local pointer into a remote pointer:

```
Permit p = new Permit();
String s = "hello";
String~ sr = s @ p;
```

Once a task has used the @ operator to create a remote pointer, it can pass the remote pointer to other tasks, which can use them until the remote pointer's permit is revoked. Operations on remote pointers perform a run-time access check: the expression "`sr.length()`" will evaluate `sr`'s length if `p` is unrevoked, and raise an exception if `p` is revoked. Permits can selectively revoke access to data: if permit `p1` is revoked while `p2` is not, then `s` is accessible through the remote pointer `(s @ p2)` but inaccessible through the remote pointer `(s @ p1)`. Note there is no way to decompose a remote pointer into its two parts: the local pointers to `s` and `p` can't be extracted from `sr`; this prevents other tasks from gaining direct access to them. In other words, a task's access to another task's data is always mediated by a permit ("complete mediation" is one cornerstone of a secure system [SS75][WBD+97]).

When a task is terminated, all the permits created by the task are revoked. This mass revocation makes all of a task's objects unreachable and therefore garbage collectable. The permits stored in the remote pointers supply sufficient information for this garbage collection; Luna does not need to keep a table of all remote references. This means that remote pointer creation is extremely fast: it consists only of pairing two words together, which requires no heap allocation, lock acquisitions, or bookkeeping.

In order to preserve the invariant that only remote pointers cross task boundaries, local pointers cannot be written to remote object fields or passed as arguments to remote method invocations, while primitive types and remote pointers can. Reads from remote objects are more flexible: a local pointer can be read from a remote object or returned from a remote method invocation, but the local pointer is automatically *promoted* to a remote pointer. This means that in the function below, the expression "`list.next`" has type `List~`, not type `List`, because from the perspective of the `second` function, a local field of a remote pointer is still remote. At run-time, "`list.next`" evaluates to a remote pointer containing the same permit as the remote pointer `list` contained. This allows a single permit to control access to the entire list, not just the first element (as would be the case with indirection-based implementations of revocation [HCC+98, Red74]). This *aggregation* of access control is crucial for mediating access to complex shared data structures; it would be unwieldy for a programmer to have to revoke every object in a large data structure individually.

```
class List {
    int i;
    List next;
}
List~ second(List~ list) {
    return list.next;
}
```

Remote pointers cannot be used interchangeably with local pointers. For instance, a hash table object expecting keys of type "`Object`" cannot be passed a remote pointer (which has type "`Object~`"). This forces the programmer to either design a new type of hash table that can accommodate remote keys (and can deal with their revocation robustly), or, more commonly, to make a local copy of a remote object's data and use that as the hash table key. Similarly, remote pointers, like Java's primitive types, must be boxed by the programmer to be placed in Java's standard container classes, such as `Vector`, that store `Objects` (this also serves a pragmatic implementation purpose, because remote pointers are a different size than local pointers, and are treated differently by the garbage collector).

These restrictions on remote pointers raise questions about Luna's expressiveness—do programmers have to write two versions of every function, one for local pointers and one for remote pointers? In fact, Luna is not usually programmed this way. Instead, Luna tasks are typically very similar to programs based on remote method invocation. In the example below, the call to `x.f` is like a remote method invocation, except that the code explicitly copies the argument and return value, rather than using an RMI-generated stub (note: the `remote` keyword is an access control keyword, like `public` or `private`; it gives other tasks the right to call the method). If desired, the remote pointer manipulation below can easily be hidden behind RMI-style interfaces.

```
class X {
   remote String f(Vector~ v) {
      Vector vlocal = new Vector(v);
      ...
      return "ok";
   }
}
X~ x = ...;
Vector~ v1 = ...;
String s = new String(x.f(v1));
```

Constructors that copy data from a remote object are common in Luna; snippets of the String constructor are shown below. Although Java has a `clone` method that makes a copy of any Cloneable object, Luna does not have an analogous "remote clone" operation for making local copies of remote objects. This is deliberate: suppose that someone passed a malicious subclass of `Vector` in as the `v` argument to the f method above. If we made a complete local copy of this object, we would have to import the code defined by the malicious subclass into our own task, in order to handle the local method invocations on the copied object. Luna *never* implicitly imports code from another task like this—this would violate the invariant that a task's threads only run the task's own code. In the example above, the expression "`new Vector(v)`" copies the data from `v`, but not the code.

```
public String(String~ value) {
  ...
  char[]~ vchars = value.chars;
  for(int i = 0; i < len; i++)
    lchars[i] = vchars[i + off];
...
}
```

Although Luna's style of "remote method invocation" is less automated than Java RMI's automatically generated stubs, it is more flexible. Programmers can delay copying data until it is needed, or use shared objects directly rather than copying.

The ability to share data between tasks raises synchronization and consistency issues: what happens if a task is terminated in the middle of an operation on shared data? Although Luna doesn't have a transaction mechanism, it allows a class to acquire a remote lock on one of its own objects for reading shared data. For writing, it's often safest to make a remote call to the task owning the object, and let that task modify its own object (if a task dies while writing to its own object, the object becomes unreachable along with the rest of the task's objects).

While remote pointers provide a revocation mechanism, they don't set a particular policy for handling revocation. It's still the application's duty to decide what to do in case of one of its communication partners becomes unreachable. Typically, well-behaved applications coordinate their actions with each other, and rely on Luna's revocation as an enforcement mechanism of last resort.

5. Implementation

This section describes the implementation of remote pointer operations in more detail. Operations on remote pointers require revocation checks, and the compiler and run-time system cooperate to implement these checks. For field and array accesses to remote pointers, the compiler emits code that enters a critical section, checks a flag in the remote pointer's permit, performs the access (or raises an exception, if the permit is revoked), and exits the critical section.

Method invocations are more intricate. A method invocation on a remote pointer calls another task's code, and therefore must execute in one of the other task's threads. This thread switch allows the caller thread to be killed without abruptly terminating the callee thread. However, Luna is implemented over Win32 kernel threads, and switching kernel threads is an expensive operation. Therefore, cross-task calls only switch kernel threads lazily, when a call must be interrupted. In the normal case, a cross-task call only switches stacks, which can be done without involving Win32. To see how this works, consider a method in task A that calls a method in task B, which in turn calls a method in task C. This sequence executes in a single kernel thread, but involves 3 separate stacks. If task B is now terminated, B's stack is deallocated, C's method continues to run in the original kernel thread, and a new kernel thread is allocated which resumes A's method (and raises an exception in A's method to indicate that the call to B aborted abnormally). This model is similar to RPC models described for the Mach[FL94] and Spring[HK93] microkernels.

In more detail, then, a cross-task method invocation enters a critical section, checks the remote pointer's permit, grabs a stack from the target task's free stack pool, exits the critical section, switches the stack pointer, pushes the arguments, and makes the call. A return from the invocation enters a critical section, returns the stack to the free pool, and exits the critical section.

At this point, the reader would be correct to worry about the cost of entering and exiting a critical section. On an 800MHz Pentium III processor, a lock followed by an unlock, both written in hand-optimized assembly using an atomic compare and exchange instruction, takes 85 cycles when measured in a tight loop. The bottleneck is access to the bus: on a multiprocessor, an atomic operation must "lock" the bus while it executes. On a uniprocessor, the bus lock may be safely omitted, which cuts the cost of a lock/unlock sequence to 21 cycles. Because the choice of uniprocessor vs. multiprocessor has such a large impact on the cost of remote pointer accesses, we report both numbers in our benchmarks. User-level thread scheduling allows further reductions of lock costs [SCM99], but we have not implemented a user-level threads package for Luna. One last optimization is possible on a uniprocessor:

remote pointer field reads (but not writes) can omit the critical section by checking the permit *after* reading the field. If the check finds that the permit is revoked, then the read is discarded, and if the check finds that the permit is valid, then the permit must also have been valid when the read occurred. Unfortunately, relaxed cache consistency prohibits this optimization on multiprocessors.

Figure 2 shows the performance of local and remote field accesses, and local and remote method invocations to a function with an empty body. Both the local and remote method invocations perform a simple method table dispatch. All measurements indicate the number of cycles taken on an 800MHz Pentium III processor with 256MB of RAM and a 256K L2 cache, measured in a tight loop (note: an empty loop takes 2 cycles per iteration; this was not subtracted from the numbers below).

	local	remote (uniprocessor)	remote (multiprocessor)
field read	3	5	96
field write	3	35	96
method invocation	11	115	238
allocate new remote pointer		2	2
allocate, revoke, and garbage collect permit		~900	~900

Figure 2: Remote pointer performance, without caching optimizations

Allocating a new remote pointer on the stack simply pairs two words (a local pointer and a permit) together on the stack; if these words are already on the stack, then the pairing operation is essentially free. Allocating a new permit, however, requires heap allocation (which accounts for over half the cost shown in the table) and some data manipulation and synchronization to maintain the permits in a tree structure for each task.

The cross-task invocations, while slower than local invocations, are nevertheless faster than round-trip IPC on the fastest x86 uniprocessor microkernels [LES+97], and are orders of magnitude faster than Win32 LRPC calls. Unfortunately, the field accesses that require locks are slow to the point of being useless: if shared data were always so expensive to access, it would make more sense to copy data than to share it. Luckily, standard *caching* and *invalidation* techniques apply to remote pointer accesses, because accesses to the data are more frequent than revocation of the data.

Caching permit information

This section describes how Luna imitates a hardware TLB to reduce the cost of repeated access checks of the same permit. However, Luna's approach differs from a hardware TLB in that a TLB operates entirely on dynamic information, while Luna takes advantage of static information to detect permit reuse. As a simple example, consider the following loop, which zeroes the elements of a remote list (using the list class defined in the previous section).

```
void zero(List~ list) {
    while(list != null) {
        list.i = 0;
        list = list.next;
    }
}
```

As Luna compiles this function from a typed high level representation to a typed low level representation, it adds type information to indicate repeated uses of the same permit. This information is computed using a straightforward intraprocedural dataflow analysis. In the `zero` loop, Luna's analysis sees that `list.next` has the same permit as `list`, and therefore every field access to `list` uses the same permit as the loop runs. The typed representation generated by the analysis is similar to the type system formalized in [HvE99]: the function is quantified over a *permit variable* ρ, and the two word remote pointer values are split into separate one word pointer and permit values, of type `List{ρ}` and `permit{ρ}`, respectively:

```
∀ρ.void zero(List{ρ} list0,
             permit{ρ} p0) {
    List{ρ} list1 = list0;
    while(list1 != null) {
        list1.i = 0;
        list1 = list1.next;
    }
}
```

This representation allows the one word pointer and permit values to be stored in unrelated stack slots and registers, and for multiple values of type `List{ρ}` to be controlled by a single permit value. Furthermore, except for the revocation check, the list traversal can treat the list traversal like a traversal of a local list, so the "`list1 = list1.next`" statement need not create a new two word remote pointer each time it executes.

A simple dataflow analysis determines that the same ρ is checked repeatedly in the inner loop. Based on this, Luna inserts code outside the loop to cache this permit's revocation flag in a register or stack slot, so that an access check is done with a simple test of the cached value, with no locking. At run-time, this caching code adds the current stack frame to a doubly linked list held by the permit, and removes the stack frame from the permit's list after the loop is finished. As a further optimization, each thread keeps recently cached permits in a small thread-local direct-mapped cache rather than releasing them immediately. The combined overhead of the pre-loop and post-loop caching code is about 30 cycles on a uniprocessor and 63 cycles on a multiprocessor if the permit is found in the thread's direct-mapped cache, and 125 cycles (uniprocessor) / 205 cycles (multiprocessor) if it is not. Thus, the optimizations pay off after about 1 to 3 accesses to the shared data. If the permit is revoked, it invalidates the cached information in each stack frame in its list by suspending each affected frame's thread, advancing the thread to a safe point (which Luna places at every call site and backwards branch), and then using the GC information at the safe point to determine which registers and stack slots must be modified to invalidate the cached access control information.

Luna optimizes further under the following conditions: (i) the loop contains no call instructions, exception handlers, or other loops, (ii) the loop uses only one permit, (iii) all paths inside the loop from the loop header block back to itself check the permit for revocation. If these are satisfied (the `zero` loop, for example, satisfies them), Luna places safe points before the loop's revocation checks rather than at the loop's backwards branches, and then omits the code for the revocation checks entirely. If the permit is revoked, and the thread is advanced to a safe point inside the loop, then the thread is left at a point where a revocation check would have appeared, and a revocation exception is raised asynchronously in the thread at this point, so that it appears to the user that there was actually a check in the code there. Luna advances Java code to a safe point by setting breakpoints rather than by polling, so this optimization reduces the per-iteration cost of the revocation checks to zero.

These optimizations preserve the original semantics of the Luna program: they only throw exceptions at points where the original Luna program could have thrown an exception, and precisely reflect the state of the program when the revocation exception is thrown. A call to `Permit.revoke` raises all the necessary exceptions before returning, so that afterwards, no threads can possibly access data using the permit (i.e. revocation is still immediate, not delayed). Stated more strongly, any possible program execution trace under these optimizations is also a legal execution trace in an unoptimized Luna implementation.

Figure 3 shows the assembly language code generated for the body of the `zero` loop. While the loop body is essentially the same as a the code generated to zero a local list, the remote traversal must manipulate the permit's linked list before and after the inner loop, and the cost of this is significant: figure 4 shows the cost of repeatedly zeroing the elements of the same list, for local and remote lists of size 10 and 100. Luna's static and dynamic optimizations make the speed of these loops reasonable, though not at good as the local case.

```
                  remote list traversal code
loop:    /* list.i = 0 (safe point)  */
         mov dword ptr [eax+8],0
         /* list = list.next (safe point)  */
         mov eax,dword ptr [eax+12]
         /* if(list == 0) goto done  */
         cmp eax,0
         je done
         jmp loop
         ...
done:
```

Figure 3: inner loop code for list traversal

list size (elements)	local	remote (uniprocessor)	remote (multiprocessor)
10	49	82	117
100	334	458	496

Figure 4: speed of local and remote list traversals

Garbage collection

GC information plays a part in caching/invalidation, and the garbage collector (an extension of Marmot's copying collector) treats remote pointers specially in other ways as well. When the collector traverses a remote pointer, it checks to see whether its permit was revoked. If the permit is unrevoked, then the remote pointer is treated as a strong pointer, while if the permit is null or revoked, the remote pointer is treated as a weak pointer, since the revocation makes the object semantically unreachable through the remote pointer. This allows a task's objects to be garbage collected even if there are outstanding revoked remote pointers to the objects. In particular, when a task is terminated, all of the task's permits are automatically revoked, causing all the task's objects to become collectable, so that a task's resources are reclaimed when the task is terminated.

6. Application 1: extensible web server

We ported a small (~4000 line) web server, originally developed for the J-Kernel, to Luna. The port changed fewer than 100 lines of existing code and added about 600 lines of new code.

The server implements Sun's servlet API [Java]. To preserve this API, the server includes wrapper classes which lazily copy remote object data into local objects when the servlet requests the data, so that a servlet need not concern itself with remote pointers directly:

```
class RemoteHttpServlet {
  Servlet servlet;
  remote HttpResponseImpl service(
    HttpRequestImpl~ req) {
      HttpResponseImpl rep = new
        HttpResponseImpl();
      servlet.service(new
        RemoteRequest(req), rep);
      return rep;
  }
  ...
}
```

The server makes a remote method invocation on the servlet's `RemoteHttpServlet` wrapper object, which wraps the request in a `RemoteRequest` object before calling the servlet's local `Servlet` object. The `RemoteRequest` constructor copies only the remote pointer to the request data, while other methods (e.g. `getContentType`) copy data from the remote request object as needed.

```
class RemoteRequest
        implements ServletRequest {
    public RemoteRequest(
      ServletRequestImpl~ req) {
      this.req = req;
    }
    public String getContentType() {
      if(ContentType == null
       && req.ContentType != null)
          ContentType = new
            String(req.ContentType);
      return ContentType;
    }
  ...
}
```

Figure 5 shows the time taken to dispatch a request to a servlet (which lives in a different task from the server) and process the response, for a servlet which returns a fixed sized message, not counting the time to transfer the data to or from the underlying sockets, and not counting the cost of spawning a thread to handle the request. The figure shows Luna's performance with and without the caching optimizations, and compares this to the performance of a modified server where both the server and servlet reside in the same task, so that no remote pointers are used. The measurements show that Luna's caching optimizations pay off for large messages, especially for a multiprocessor.

response size (bytes)	Single task (local pointers only)	Multiple tasks,no caching (uni/multi-processor)	Multiple tasks, caching (uni/multi-processor)
100	233μs	266μs / 270μs	266μs / 266μs
1000	300μs	400μs / 467μs	333μs / 333μs
10000	1000μs	1500μs / 2403μs	1070μs / 1137μs

Figure 5: servlet response speed

7. Application 2: active cache

The web server in the previous section used very high-level RMI-style communication between the server and its servlets. This section presents an extension to the popular Squid web cache [Squ] to support active content analysis and generation, built using a lower-level shared memory style of communication between the Squid cache and the Luna active extensions. We choose a low-level style partly for performance, and partly because this interfaces naturally with Squid (which is written in C, not Java). A low-level interface is not always convenient to use, but it would not be hard to build a higher-level interface, such as the Active Cache Protocol [CZB98] over the lower level. The low-level interface also meant that there were only about 400 lines of Luna code, compared to 60,000 lines of Squid C code.

The active cache consists of Squid, one "root task" written in Luna, and an arbitrary number of extension tasks written in Luna. If an HTTP request hits in Squid's cache, and the cache entry is marked as belonging to an extension, then Squid selects a thread from the root task's pool of waiting threads, and passes the request to the thread. The root task then makes a cross-task call into the proper extension, which in turn calls the root task one or more times to read data from disk or send data out a socket.

Squid must pass the original request data into the root task and then on into the extension task. Similarly, the extension task forms a response, which it passes back to the root task and ultimately into C code that writes the response to the socket. To avoid copying the data, the C code allocates a large pool of fixed-sized buffers. Each buffer contains a proper Java header to make it look like a Java object of type `byte[]`. However, the buffers do not live in the Java heap and are not garbage collected. This raises a delicate issue: if the C code wants to reuse or deallocate a buffer, how can it be sure that there are no Java pointers to it?

For example, suppose that a buffer received an HTTP request for extension A, and this buffer is later overwritten with an HTTP request for extension B. A should not be able to use its pointer to the buffer to read B's data. To prevent this, the root task protects the buffers with permits, and revokes access to a buffer before giving the buffer to another extension task. Similarly, B should not be able to read A's old data in the buffer. Therefore, the C code sets the "length" field of the buffer's Java header to exactly the size of B's received data, so that none of A's old data is visible from Java (because of Java's bounds checking, which Marmot implements using a combination of static analysis and run-time checks).

Since the buffers live in the C heap, they aren't automatically "charged" to the task that is using them at any given moment. This is a limitation of Luna's simple task model, but it wouldn't be hard to work around: Squid would simply have to track the buffer usage itself, and bill the appropriate task. The extensions presumably trust Squid to generate an accurate bill. Rather than attempt to stretch the task model to cover all possible trust/sharing/billing models, it seems simpler to use tasks as a default resource accounting model, and have trusted tasks account for resources that fall outside the task model.

Using the Pentium's cycle counter, we measured the total time for each stage of a request, where the request is about 370 bytes long, and the extension sends a preallocated 500 byte buffer back as a response. The dominant costs are in Squid and the C socket routines: it takes 6397μs for Squid to read and parse request and see that it is a cache hit, and it takes 5930μs to write the final response to a C socket. By contrast, the cost of the Luna code is small (in part because no data copying is required): 4μs to prepare the request data for the extension and signal a thread in the thread pool, 12μs to context switch to the new thread, and 57μs for the root task and extension code to generate and deliver the response to the C socket routines.

8. Comparison to related work

SPIN [BSP+95] used Modula-3 to safely extend an operating system kernel. It demonstrated that a high level language could eloquently express the interfaces between layers and enforce the layers' abstractions. Luna maintains both these properties: although the tilde throws in an additional constraint, Luna still expresses interfaces in the language itself, and uses the language features to enforce abstractions.

Several projects [FFK+99] [HCC+98] [VB99] [BHL00] [BTS+00] [RCW01] have explored the idea of safe language tasks. Rudys et. al. [RCW01] propose a model where a task is defined by only code and threads, and object ownership is not tracked. Alta [BTS+00] restricts the types of objects available for inter-task sharing to some extent, but otherwise provides an all-or-nothing choice: either unrestricted sharing between two tasks or no sharing at all. Luna tries for the best of both worlds: allow sharing between arbitrary tasks, but use the type system to track the sharing.

KaffeOS [BHL00] uses write barriers to restrict the propagation of inter-task pointers, so that sharing between tasks is confined to specially allocated buffers (it severely restricts the types of objects that may be placed in the buffers: no abstract datatypes with overridden methods, for example). KaffeOS doesn't have a static distinction between local and remote pointers, so the write barrier cost is incurred for all pointers, and errors are caught at run-time rather than statically. On the other hand, KaffeOS requires only changes to Java's implementation, not the Java language.

Although this paper discusses implementation issues directly related to remote pointers, there are other important implementation issues. KaffeOS, for example, garbage collects different tasks separately to minimize the interference between tasks (with respect to memory accounting and predictable GC behavior), while Luna still only uses a single collector. MVM [CD01] and ShMVM [CDN02], in addition to dealing with separate garbage collection, attempt to share code transparently between tasks to a much greater extent than Luna does, and provide a mechanism for each task to load isolated native code. We believe that the techniques from these systems could be integrated into Luna without much difficulty, although there is one difficult time/space tradeoff: greater sharing of compiled code between tasks would save memory but limit Luna's whole-task optimizations.

Both KaffeOS and Alta concentrate on shared memory communication; while Alta supports inter-task RPC, it is more than an order of magnitude slower than Luna's cross-task calls. The J-Kernel [HCC+98], on the other side, supports RPC (with deep copies of arguments and return values) but not shared memory. The J-Kernel's copy routine is trusted and not customizable, whereas Luna lets the user write their own copy routines by using field accesses to remote objects (such as the lazy copies implemented in our web server). A drawback to Luna's approach is that such customization demands serious optimization to prevent excessive revocation checking overhead.

Luna's remote pointer types and the lower level annotated typed representation were inspired by the region annotations of Tofte et al [TT94][CWM99]. Originally, we imagined having one "region" per task, but then decided to let a task allocate multiple "permits", so that it can selectively revoke some permits but not others.

9. Conclusions

When we first started working on language based protection, we had wild hopes of outperforming traditional operating systems and microkernels. The lack of separate address spaces would allow fine-grained sharing and make protection and communication dirt cheap. All that we needed to do was to solve the (apparently) minor problems of revocation, resource control, and terminating tasks so that the system would be robust enough to handle servers, agents systems, and active network systems. We found that these issues weren't nearly as easy to solve in a tightly coupled safe-language system as they were in a traditional system that clearly divides tasks into separate address spaces (and has a very fast TLB enforcing this division). Rather than admit defeat to traditional systems on these issues, we introduced a clear separation between tasks into Luna, but unlike the dynamic separation based on virtual memory address spaces, we made a static distinction in the type system. The contribution of this paper lies in:

- an argument for a *task* model in safe language setting, where each task has its own code, objects, and threads
- a type system extension (~) which allows complete mediation of inter-task communication
- a run-time mechanism that uses the ~ to implement revocation and termination. We found that the costs of revocation were reasonable for method invocations, but array and field accesses were challenging to optimize, requiring extensive cooperation between the compiler and run-time system.
- the application of these ideas to a real-world language, using the fastest optimizing Java virtual machine (Marmot) that we could get our hands on, to demonstrate that our mechanism did not preclude global optimizations or optimization of operations on local pointers, even in the presence of dynamic loading
- the demonstration of a variety of approaches to inter-task sharing (shared-data, remote procedure call, lazy copying) in two extensible applications

The result is a system that combines the robustness, structure, and communication flexibility of a microkernel with the ease of expression, portability, and abstraction enforcement of a high-level programming language.

References

[BH99] G. Back and W. Hsieh. *Drawing the Red Line in Java.* Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems, Rio Rico, AZ, March 1999

[BHL00] G. Back, W. C. Hsieh, and J. Lepreau. *Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java.* Proceedings of the 4th Symposium on Operating Systems Design and Implementation, October 2000

[BR00] C. Bryce and C. Razafimahefa. *An Approach to Safe Object Sharing.* ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Minneapolis, USA, October 2000

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System.* 15th ACM Symposium on Operating Systems Principles, p. 267–284, Copper Mountain, CO, December 1995.

[BTS+00] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. *Techniques for the Design of Java Operating Systems.* Proceedings of the USENIX 2000 Annual Technical Conference, San Diego, CA, June 2000

[CD01] G. Czajkowski and L. Daynès. *Multitasking without Compromise: A Virtual Machine Evolution.* In OOPSLA 2001, Tampa Bay, FL, Oct. 2001.

[CDN02] G. Czajkowski, L. Daynès, and N. Nystrom. *Code Sharing Among Virtual Machines*. In ECOOP 2002, Málaga, Spain, Jun. 2002.

[CWM99] K. Crary, D. Walker, and G. Morrisett. *Typed Memory Management in a Calculus of Capabilities*. In 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages

[CZB98] P. Cao, J. Zhang, and K. Beach. *Active Cache: Caching Dynamic Contents on the Web*. Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98), pp. 373-388.

[DP93] P. Druschel and L. L. Peterson. *Fbufs: A high-bandwidth cross-domain transfer facility*. Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, p. 189-202, Dec. 1993.

[FFK+99] M. Flatt, R. B. Findler, S. Krishnamurthi and M. Felleisen. *Programming Languages as Operating Systems (or, Revenge of the Son of the Lisp Machine)*. International Conference on Functional Programming (ICFP), Paris, France, Sep. 1999.

[FKR+99] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgard, D. Tarditi. Marmot: An Optimizing Compiler for Java. Microsoft Research Technical Report MSR-TR-99-33, June 1999.

[FL94] B. Ford and J. Lepreau. *Evolving Mach 3.0 to a Migrating Thread Model*. Proceedings of the Winter Usenix Conference, January 1994.

[Gen] General Magic. Odyssey. http://www.genmagic.com/agents.

[GMS+98] M.Godfrey, T.Mayr, P.Seshadri, and T. von Eicken. *Secure and Portable Database Extensibility*. Proceedings of the 1998 ACM-SIGMOD Conference on the Management of Data, p. 390-401, Seattle, WA, June 1998.

[HCC+98] C. Hawblitzel, C. C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. *Implementing Multiple Protection Domains in Java*. 1998 USENIX Annual Technical Conference, p. 259-270, New Orleans, LA, June 1998.

[HK93] G. Hamilton and P. Kougiouris. *The Spring Nucleus: a Microkernel for objects*. Proceedings of the Summer 1993 USENIX Conference, p. 147-159, Cincinnati, OH, June 1993.

[HLP98] R. Harper, P. Lee, and F. Pfenning. *The Fox Project: Advanced Language Technology for Extensible Systems*. Technical Report CMU-CS-98-107, Carnegie Mellon University.

[HvE99] C. Hawblitzel and T. von Eicken. *Type System Support for Dynamic Revocation*. ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta, GA, May 1999.

[Java] JavaSoft. *Java Servlet API*. http://java.sun.com.

[Javb] JavaSoft. *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?* JDK 1.2 API documentation, http://java.sun.com.

[LES+97] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, T. Jaeger. *Achieved IPC Performance*. 6th Workshop on Hot Topics in Operating Systems, Chatham, MA, May 1997.

[MWC+98] G. Morrisett, D. Walker, K. Crary, and N. Glew. *From System F to Typed Assembly Language*. 25th ACM Symposium on Principles of Programming Languages. San Diego, CA, January 1998.

[NL98] G. Necula and P. Lee. *The Design and Implementation of a Certifying Compiler*. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation. Montreal, Canada, June 1998.

[RCW01] A. Rudys, J. Clements, and D. S. Wallach. *Termination in Language-Based Systems*. Proceedings of the Network and Distributed Systems Security Symposium (NDSS '01), San Diego, California, February 2001.

[Red74] D. D. Redell. *Naming and Protection in Extendible Operating Systems*. Technical Report 140, Project MAC, MIT 1974.

[SCM99] O. Shivers, J. W. Clark and R. McGrath. *Atomic heap transactions and fine-grain interrupts*. Proceedings of the 1999 ACM International Conference on Functional Programming (ICFP), Sep. 1999, Paris, France.

[Sha97] Z. Shao. *Typed Common Intermediate Format*. 1997 USENIX Conference on Domain-Specific Languages, Santa Barbara, California, Oct. 1997.

[Squ] Squid Web Proxy Cache, www.squid-cache.org.

[SS75] J. H. Saltzer and M. Schroeder. *The Protection of Information in Computer System.* Proceedings of the IEEE, Volume 63, Number 9, p. 1278–1308, September 1975.

[TT94] M. Tofte and J.P. Talpin. *Implementation of the Typed Call-by-Value Lambda Calculus using a Stack of Regions.* 21st ACM Symposium on Principles of Programming Languages, p. 188–201, Portland, OR, January 1994.

[VB99] J. Vitek and C. Bryce. *The JavaSeal mobile agent kernel.* In First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA'99), October 1999.

[WBD+97] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. *Extensible Security Architectures for Java.* 16th ACM Symposium on Operating Systems Principles, p. 116–128, Saint-Malo, France, October 1997.

[WLH81] W. A. Wulf, R. Levin, and S. P. Harbsion. *Hydra/C.mmp:An Experimental Computer System.* McGraw-Hill, 1981.

[WGT98] D. Wetherall, J. Guttag, and D. L. Tennenhouse. *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols.* IEEE OPENARCH'98, San Francisco, CA, April 1998.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

❖ Atos Origin B.V. ❖ Freshwater Software ❖
❖ Interhack Corporation ❖ Microsoft Research ❖
❖ Motorola Australia Software Centre ❖ OSDN ❖
❖ Sendmail, Inc. ❖ Sun Microsystems, Inc. ❖
❖ Sybase, Inc. ❖ Taos: The Sys Admin Company ❖
❖ UUNET Technologies, Inc. ❖ Ximian, Inc. ❖

SAGE Supporting Members

❖ Certainty Solutions ❖ Collective Technologies ❖
❖ ESM Services, Inc. ❖ Freshwater Software ❖
❖ Microsoft Research ❖ OSDN ❖ Ripe NCC ❖

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-06-4